The Pragmatic Bookshelf

TURING

The Passionate Programmer

Creating A Remarkable Career In Software Development

我编程，我快乐

程序员职业规划之道

[美] Chad Fowler 著
于梦瑄 译



- 程序员生存、成功、制胜的法则
- 源自IT精英的职业发展秘诀
- 热爱工作，享受生活

人民邮电出版社
POSTS & TELECOM PRESS

“如果你对软件开发这行有浓厚的兴趣，如果你想成为一名出色的软件开发者，如果你想每天充满激情地工作，想要把开发软件视为一项事业而不仅仅是一份工作，那就一定要读完这本书。Chad Fowler在书中与我们分享了实用有效的探索式方法、经验和态度，告诉你如何尊重并热爱你的职业，助你成为行业中的佼佼者。”

——Robert Martin, Object Mentor公司总裁

“这绝对是本好书，Chad Fowler用聊天的形式提供了很多职业规划方面的建议。这些建议帮助程序员改变工作习惯，让人带着满腔热情走向职业的辉煌。”

——亚马逊读者评论

The Passionate Programmer

Creating A Remarkable Career In Software Development

我编程，我快乐

程序员职业规划之道

要在当今的IT职场取得成功，必须像经营企业那样对待你的事业。在本书中，你将学到如何规划自己的职业生涯，让它向着你选择的目标前进，使人生更快乐、更美好。

作者运用其独特的思维方式，启发程序员不能只注重技能上的提高，还要关注自己的职业发展。书中涉及新旧技术的取舍、技术与业务的关系、技术是要专精还是要广博等，相信这也是长久以来困扰你的问题。带着这些问题去阅读此书，定会受益良多。

此外，本书中的每一章都包含一篇或几篇各领域成功人士的文章，让你直接了解他们如何规划自己的职业生涯！

The Pragmatic Bookshelf

图灵网站：www.turingbook.com　热线：(010)51095186
反馈/投稿/推荐信箱：contact@turingbook.com
有奖勘误：debug@turingbook.com

分类建议　计算机/程序设计

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-23352-3

9 787115 233523

ISBN 978-7-115-23352-3
定价:39.00元

TURING

The Passionate Programmer

Creating A Remarkable Career In Software Development

我编程，我快乐

程序员职业规划之道

[美] Chad Fowler 著
于梦瑄 译



人 民 邮 电 出 版 社
北 京

图书在版编目（CIP）数据

我编程，我快乐 ：程序员职业规划之道 / （美）福勒（Fowler,C.）著 ；于梦瑄译. -- 北京 ：人民邮电出版社，2010.8
ISBN 978-7-115-23352-3

Ⅰ. ①我… Ⅱ. ①福… ②于… Ⅲ. ①程序设计—工作—基本知识 Ⅳ. ①TP311.1

中国版本图书馆CIP数据核字(2010)第129402号

内 容 提 要

本书讲述程序员的职业规划之道——如何规划职业生涯，如何按照自己选择的方向发展职业，如何沿着你构建和销售自己产品的路径一步步地实现自己的职业目标。全书共5章，涉及如何挑选职业发展的技术和商业领域、在经济飞速发展的今天应具备哪些技术、如何为公司创造价值以及如何推销自己。

本书适合所有行业的软件开发人员阅读。

我编程，我快乐：
程序员职业规划之道

◆ 著 [美] Chad Fowler
译 于梦瑄
责任编辑 傅志红
执行编辑 丁晓昀 罗 婧

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号
邮编 100061 电子函件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京隆昌伟业印刷有限公司印刷

◆ 开本：800×1000 1/16
印张：14
字数：185千字 2010年8月第1版
印数：1 – 4 000册 2010年8月北京第1次印刷

著作权合同登记号 图字：01-2010-1843号

ISBN 978-7-115-23352-3

定价：39.00元

读者服务热线：(010)51095186 印装质量热线：(010)67129223
反盗版热线：(010)67171154

版 权 声 明

读者感言

如果你对软件开发这行有浓厚的兴趣，如果你想成为一名出色的软件开发者，如果你想每天充满激情地工作，想要把开发软件视为一项事业而不仅仅是一份工作，那就一定要读完这本书。作者 Chad Fowler 在书中分享了实用有效的探索式方法、经验和态度，告诉你如何尊重并热爱你的职业，助你成为行业中的佼佼者。

——Bob Martin, Object Mentor 公司总裁

这本书最棒的地方在于它非常符合我的实际情况，并且为我提供了很多可行的计划。通过阅读这本书，我发现我的现状并不可怕，很多人都和我一样。它告诉我现在能做什么，明天要做什么，以及我未来的职业道路应该怎样规划。

——Kent Beck，程序员

6 个月前，我正打算换工作，就在这个时候我读到了这本书。去年 11 月到今年 5 月的一些经历让我最终决定继续从事软件开发工作，而且我还要充满激情地去工作，争取成为行业佼佼者。摆在你面前的这本书，为你提供了很多有益的建议，它能够激发你的热情，是你实现目标的指南针。

——Sammy Larbi, Codeodor.com 首席 Spaghetti 程序员

序

每个人都有卓越之处，但需要找到自己真正喜欢的事情把它激发出来。如果你不喜欢周围的环境，不喜欢你使用的工具，不喜欢工作的领域，那你的卓越之处又怎么可能被激发出来呢？

在加入 37signals 和开发 Ruby on Rails 之前，我做过很多工作，当然那些都称不上卓越。我虚度光阴，过一天算一天。6 个月后，我才发现事情的严重性——我一事无成。

我很懊悔。世界不会因我没有完成工作而停滞不前，我讨厌这种感觉，这让我觉得我的存在毫无意义。如果你想变得卓越出众，就必须相信你所做的事情正在推动整个世界的发展。

那时候我在工作上毫无成就，生活也因此受到了影响。我觉得我的工作没有意义，固而很难鼓起勇气努力让它变得有意义。

我认为从事卓越的职业是拥有有意义的人生的起点。你不仅成为了一名更优秀更具价值的员工，更成为了一个更有价值的人。

这就是这本书的重要之处。它不是教你如何做出一个更好的小程序并体会到工作的稳定感，而是帮助你成就更卓越、更有意义的人生，工作只是这其中的一部分。

——David Heinemeier Hansson

Ruby on Rails 之父，37signals 合伙人

前　言

本书能帮助你找到事业上的快乐感和满足感。快乐和满足并不总源自机遇。你需要思考，确定目标，然后行动。犯了错误，你要想怎么去改变它。本书为你在软件开发行业走出一条成功的职业发展道路提供了策略，进而助你成就成功的人生。

本书激励人们树立远大理想，追求一个卓越的人生抱有期望。当我们开始职业生涯的时候，出发点并不是追求卓越的人生。大部分人都会受媒体、以及朋友、熟人和家人的影响，随波逐流，降低了对自己的期望。所以，虽然追求卓越的人生是理所当然的，但却又不是显而易见的，你必须要去发掘它。

一个成年人大部分的清醒时间都在工作中度过。2006 年美国劳工统计局统计数据[①]显示，美国人平均花在工作上的时间占他们所有清醒时间的一半，而休闲和运动的时间只占 15%。事实证明，我们的工作基本上就是我们的生活。

如果生活的大部分时间都被工作占据着，那么热爱工作就是热爱生活。比起那些枯燥的简单任务，充满挑战、有驱动力、有回报的工作更能让你有动力在清晨从温暖的被窝里爬起来。工作做得好意味着你在充分发挥着才能。相反，如果工作做得不好，就证明你大部分时间都只能在懊悔，懊悔自己碌碌无为。

---

① http://www.bls.gov/tus/charts/

我们最终的目标是追求快乐。一旦满足生存的基本需求，人们就会转而去追求快乐。然而，我们的行动往往与此目标相悖。这是因为人们往往纠缠于做某事的方法，而忘记了最终的目的。

如果我有更多的钱，或许我会更快乐；如果我的成就被认可，或许我会更快乐；如果我升职或者有名望了，或许我会更快乐。但是如果我贫穷而且做着琐碎平凡的工作，我会快乐吗？如果可能，我应该追求更高的薪水，还是更好的工作呢？

或许我不会快乐。但可以肯定的是，当我们以追求快乐为核心目标，并且以此作为首要的推动手段时，那在追求目标的过程中，就会更正确地做出每一步的选择。更高的薪水或许是值得追求的，或许能带给你快乐。但是如果你把目光放远一些，你就会发现自己为了追求更高的薪水，或许就会失去了快乐。这听起来有些荒谬，但是我经历过，你也一样可能会经历。

我希望我的建议可以帮助你成就更加快乐更加有回报的职业生涯，进而成就更快乐的人生。遵循这些建议，或许你的荷包会更加丰盈，或许你会得到更多的认可甚至是成名。但是，请不要忘记，这些都不是最终目标，它们只是实现最终目标的方法。

不要害怕失败

在我创造卓越职业生涯的道路上，撰写此书的第一版是重要的一步。这本书的第一版起名为 *My Job Went to India (And All I Got Was This Lousy Book): 52 Ways to Save Your Job*。书的封面图片很有趣：一个人举着一个标志牌，上面写着“要编码，要吃饭。”书名和书皮耀眼的红色是用来突出西方世界的恐惧——他们的工作将被外包给那些国外廉价的编程团队。

这个设计的问题在于用错了封面图片。如果你只是想要“留住”你的工作，我无能为力。我不是在教你如何继续平庸着、挣扎着留住工作不被炒鱿

鱼。我要告诉你，你要出色，你要赢。就像在赛跑中，你要总想着怎么不输，那肯定不会赢得比赛。同样，总想着怎么避免糟糕地活着，那你也不可能成为生活中的赢家。任何人都不应该时刻想着如何避免失败。

我决定要让职业生涯变得卓越的那一刻，至今仍记忆犹新。我轻松地完成高中和大学学业，然后成为了一名平庸的专业萨克斯演奏者。我的天赋，再加上机遇，使我取得了一些成功，让我在一家世界知名的公司中找到了一个显要的技术职位，收入颇丰。但是我很清楚自己只是为了生活。

一天晚上下班后，我走进了一家书店，碰巧在推荐新书的书架上看到Kent Beck 写的 *Extreme Programming Explained*①[Bec00]一书。书的副标题是 *Embrace Change*（拥抱变化）。我喜欢“变化”这个观点。我能集中注意力去干一件事的时间很短。那时候，我一直在频繁地换工作。“软件开发方法论”这个观点听起来极度无聊，而且有些偏向管理。但是，我认为如果这其中包含很多变化，那就可能帮助我改进工作而不是老想着换工作。

事实证明挑选这本书是个明智的选择。我对它爱不释手，一口气读完了全部内容。我在网上搜索所有关于极限编程（XP）的内容，深深地被这些观点所打动。我找到公司的 CIO，试图让他接受这个观点。最终，CIO 和所有员工都被说服了，采纳了极限编程这一概念，还指派我们一组人参加 Object Mentor 的极限编程培训课程。

如果想要学习极限编程，就来参加极限编程俱乐部（Extreme Programming Immersion）。这就像最受欢迎的摇滚巨星进行为期一周的演出，而我可以进入后台与他们近距离接触。和这些人相处，我变得更加明智，更具创造力。课程结束后，一想到自己要回到办公室的小格子间，继续碌碌无为地工作，我就非常失落。

① 本书中文版《解析极限编程：拥抱变化》已经由电子工业出版社出版。——编者注

我的同事 Steve（本书收纳了他写的一篇文章）和我都认为，与这些人尽可能保持密切接触的唯一方法就是成为他们中的一员。也就是说，如果我想和那些能让我进步的人在一起，就得成为他们中的一员，而在某个公司工作或者在大学进修是无法让我走向卓越的。我要做的是弄明白成为他们中的一员意味着什么，然后努力去做。所以我向 Steve 宣布，我要成为他们中的一员。

这就是我职业道路上的转折点。多年后，要不是 Steve 提起，我都忘了我们俩的那次交谈。我当时告诉他第一次有人邀请我在大会上做重要演讲。我自己都不敢相信，居然会有人邀请我在关于软件的会议上做重要发言。我——终于如愿以偿——成为了他们当中的一员。

之前我从没有接受过任何计算机程序的相关培训。做程序员之前，我是搞音乐的，大学学的也是音乐。大学文凭对搞音乐的人来讲并不重要，所以与音乐无关的课我一律不上。虽然毕业前我的学分多到可以拿到任何学位，但是因为有些必修课没有上，所以不得不又多上了几年才毕业。要成为一名专业软件开发人员我不够资格，至少从招聘广告上的要求来说，我不符合传统意义上对一名普通软件工程师的要求。

尽管我不够资格成为传统意义上的普通软件开发人员，但是我从乐器演奏者的工作经历中领悟到很重要的一点，使我最终跨过普通软件开发人员的这一步（其实，谁又满足于做一名普通的软件开发人员呢？）。想要找一份稳定安逸工作的音乐人成不了音乐家。音乐这行很残酷，压根也不可能安逸。能成为专业乐手的人都想成名。一名乐手要不就努力想成名，要不就根本别踏进这一行。

经常有人问我为什么很多优秀的乐手同时又是很棒的软件开发工程师。这绝对不是因为这两种工作调动的大脑功能一样，也不是因为它们都是追求细节和创造力的工作，而是因为一个渴望成功的人肯定要比那些只是单纯完

成工作的人更有可能成功。即使我们不能成为 Martin Fowler、Linus Torvalds 那样的人，但确定高目标至少可以让我们不再平凡。

制订自己的计划

很多人总是忽略自己的计划，一味地跟随别人的计划。为了有别于他人，你应该停下来仔细审视自己的职业。不要去跟在别人的计划后面跑，你应该按着自己的计划发展。

那么，这个计划又该怎么制订呢？软件是一门生意。作为软件开发人员，我们就是生意人。公司雇我们，绝不是因为他们爱我们。事实上，他们以前从没爱过我们，将来也绝不会。否则软件就不是一门生意了。做生意可不是说让我们每天有个地方去，有事干；做生意的目的就是盈利。要想在公司中成为佼佼者，就必须要懂得如何将自己融入这门生意，然后创造利润。

稍后我会提到，长期雇用一个人对公司来讲是一笔很大的开销。公司雇用你，是在你身上投资，而你要做的就是让公司的投资得到回报。要评定自身的表现，就要看你能给你的老板创造多少商业价值。

把你的职业想象成你正在制作的产品的生命周期，你的技术成就了这个产品。本书中，我们要讨论在设计、生产及销售一件商品时，应该注意的四方面内容，以及如何将这四方面内容运用于职业发展。

- 选择市场。一定要谨慎地挑选你要关注的技术和商业领域。如何权衡风险和收益？供需关系又会如何影响你的决定？
- 投资。你的知识和技术是这件商品的基础。要在这两方面合理地投资，这是市场化的重要前提。只知道如何在理论上使用 VB 或者 Java 已经远远不够了。那么在新的经济环境下，又有哪些技术是你应该具备的呢？

- 执行。单纯依赖技术出色的员工，并不能给公司带来利益。员工必须要有产出。那作为员工，你又如何在不把一切搞得一团糟的前提下，更好地产出呢？你又怎么能知道自己是不是在为公司创造最有利的价值呢？
- 市场！即使是市面上有史以来最好的产品，若是无人知晓，又怎么会有人来购买呢？你怎样做才能在公司和行业中得到认可，但又不必阿谀奉承呢？

新版

本书是 *My Job Went to India (And All I Got Was This Lousy Book): 52 Ways to Save Your Job* 的第二版。撰写这一版是为了进一步强调创作前一版本的真实目的——创造卓越的职业生涯。在这一版中，我不仅换了一个更加明确的书名，也添加了新的内容。

David Heinemeir Hansson 是 Ruby on Rails 之父以及 37signals 的合伙人，他为本书写了新的序。

书中每一章都包含一篇或几篇别人撰写的文章，作者或者与我相识，或者与我一起工作过，他们的职业生涯都是非常卓越的。这些发明家、开发者、管理者和企业家在通往成功的道路上做出过很多决定，这些文章可以让读者对他们做出决定的过程有深入的了解。这些文章也足以证明，本书介绍的技巧并非是只适用于完美环境的理想化的建议，它们是真实存在的，是任何人都可以掌握并且做到的。

此版本中删除了前一版本中的某些内容，又增加了一些新的内容。老版中的最后一章"*If You Can't Beat 'Em*"被全部删除。新的内容贯穿于全书，这些新的知识是在第一版出版后我学到的。

第一版中的某些内容被重新编排进了此版本中的“练习”中。

我们更新了上一版的前言和结尾，目的是为了更加突出本书的主题，即创造卓越的职业生涯。

本书的目的是提供一套系统的方法来指导读者创造卓越的软件开发职业生涯。书中，我们将讲述具体的事例，还会提供即时可行的练习。不管是从近期看，还是从长远看，它们都会产生积极的作用。

如前所述，我们不会谈论如何留住你的工作。如果你正在担心要失去工作，那么踏上创造卓越职业生涯之路，会帮助你消除那些恐惧。卓越的软件开发人员绝对不会萎靡不振，他们不会毫无成果地寻找工作。所以，你也不要担心。请你相信自己一定会成功，这样就不会感到恐惧了。

致　谢

首先我要感谢Dave Thomas和Andy Hunt，这本书是为他们而写的。*The Pragmatic Programmer* [①][HT00]这本书就像是催化剂，至今仍鼓舞着我。多亏了Dave的鼓励和指导，我才有足够的信心来写这本书。

Susannah Pfalzer是本书第二版的编辑。我所指的“编辑”工作的内容包括推动、鼓励、支持、促进，当然还有编辑。她很有耐心，而且总能用恰当的言辞激励我，而不是威胁吓唬我让我感到害怕而逃避，这正是我完成这本书所需要的。如果没有 Susannah，这本书到现在可能还只是杂乱无章的构思。

David Heinemeier Hansson为本书作序。他是37signals的合伙人以及Rails之父，他的职业经历正是本书观点的有力佐证。此外，在职业生涯中，我认识了很多优秀的人，他们也为此书的创作提供了灵感，非常感谢他们激励了我和我的读者，他们是：Stephen Akers、James Duncan Davidson、Vik Chadha、Mike Clark、Patrick Collison和Tom Preston-Werner。

很多审稿人为本书的第二版草稿提供了非常有价值的反馈。草稿中有一章开始时安排得极不妥当，而出色的审稿人竟可以把它梳理得井井有条。感谢Sammy Larbi、Bryan Dyck、Bob Martin、Kent Beck、Alan Francis、Jared Richardson、Rich Downie和Erik Kastner。

Juliet Thomas是本书第一版的编辑。她的热忱和独到的观点非常可贵。审稿人对本书第一版反馈了大量建议。这些建议使本书更加完善，对他们花

① 本书中文版《程序员修炼之道——从小工到专家》已经由电子工业出版社出版。——编者注

费的时间、精力以及提出的建议，我感激不尽。他们是：Carey Boaz、Karl Brophey、Brandon Campbell、Vik Chadha、Mauro Cicio、Mark Donoghue、Pat Eyler、Ben Goodwin、Jacob Harris、Adam Keys、Steve Morris、Bill Nall、Wesley Reisz、Avik Sengupta、Kent Spillner、Sandesh Tattitali、Craig Utley、Greg Vaughn和Peter W. A.Wood。

这些年，我有幸和很多优秀的伙伴共事，是他们为本书的创作提供了灵感，感谢他们的倾听、教导和交流。他们是Donnie Webb、Ken Smith、Walter Hoehn、James McMurry、Carey Boaz、David Alan Black、Mike Clark、Nicole Clark、Vik Chadha、Avi Bryant、Rich Kilmer、Steve Akers、Mark Gardener、Ryan Ownens、Tom Copeland、Dave Craine、John Athayde、Marcel Molina、Erik Kastner、Bruce Williams、David Heinemeier Hansson、Ali Sareea和Jim Weirich。

感谢我的父母对我一贯的支持。最后还要感谢我的太太凯利，她让这一切都有了意义。

目　录

第 1 章

选择市场



你马上就要进行一次大的投资，也许并不是要投入大笔金钱，而是时间，是你的一生。大都数人对待工作的态度往往都是顺其自然，走一步看一步——我们刚刚深入了解了 Java 或者 VB，老板有一天突然参加了一个热门技术的培训，于是我们就转而学习新技术，直到有人又把新的东西递到我们手里。我们的职业道路就是由一连串没有方向的偶然构成的。

在《程序员修炼之道》一书中，Dave Thomas 和 Andy Hunt 谈到了编程中的偶然性。下面这个场景，会引起大部分编程员的共鸣：当你开始做一个程序的时候，或许手头上有一个从网上复制的示例程序，看上去这个程序可以使用。为了满足你的需要，你会对这个程序稍加改动——添加一些代码，再加一点。你根本就不知道自己在做什么，只是不断地做一些小的修改，直到这个程序完全满足你的需要。但问题是，这样做就像是用纸牌搭建房子，每增添一张纸牌，就增加了一分纸房子坍塌的危险。你根本就不知道这个程序是如何工作的，所以你每做一点儿改动，都有可能导致你的程序完全失败。

作为软件开发人员，用这种投机取巧的方式来编程显然不是什么好主意。但是很多人正是让偶然来决定职业道路上的各种选择。我们应该在哪种技术上投资？应该专注于哪个领域？是应该扩展知识面，还是深入学习一门学问？这些问题都是值得我们细细斟酌的。

想象一下你开了一家公司，现在正要生产你们的明星产品。如果这个产品失败了，公司就会破产。你会花多少精力来思考此产品的消费者是谁？在产品进入生产流程之前，你又会用多少时间来弄明白这个产品到底是什么？我相信你肯定会仔仔细细地考虑其中的每个小细节，然后亲自做出决定。

但是，在职业道路上，面临选择的时候，我们为什么就缺少了这番心思呢？如果你把自己的职业当成是一门生意（事实上它就是一门生意），那么你的“产品”就是由你提供的服务构成的。这些服务是什么？你又会把它们

出售给谁？接下来的一年，对此种商品的需求是会增加还是减少呢？在这些选择上你愿意投下多少赌注？

读完本章的内容后，你会找到答案。

1 稳定成熟的技术还是未成熟的新技术?

如果你想投资，可以有许多方法。你可以把钱存进银行，但是利息的增长往往跟不上通货膨胀的速度。买国债也是个办法，但同样，收益也不会很高。不过，这两种投资方法都无需承担什么风险。

你也可以选择把钱投入一个小规模的创业公司。投入几千美金换取公司的一小部分股份。如果公司的决策正确，而且这个决策被有效地执行了，那么你就有可能挣一大笔钱，否则就有可能会血本无归。

风险收益平衡不是什么新概念。小时候玩追人游戏的时候，如果我一直不停地跑到中间，大家都会觉得吃惊，但这样做就没人能追到我。这一概念充斥在我们的日常生活中。你要去参加一个会议，可已经迟到了，在考虑如何选择一条最快的路线时，就用到了风险收益平衡。你会想，如果交通畅通，我从第 32 大街走的话，就可以提前 15 分钟到；如果交通拥堵，我就彻底没希望了。

在有目的地选择投资哪种技术和领域时，风险收益平衡是一个很重要的权衡因素。15 年前，学会如何用 COBOL 编程是一项低风险的投资。那个时候，COBOL 程序员的竞争很激烈，平均工资并不高。掌握这门技术，你很容易就可以找到工作，但这份工作的经济回报较低。这就是低风险，低回报。

同样在那个时期，如果你选择学习了 Sun 公司的新语言 Java，或许你不能轻易找到工作，因为那时候使用 Java 编程的公司很少。谁都不知道 Java 到底能用来做什么。

但是如果在那一时期你仔细观察这个行业，就像 Sun 公司一样，你或许会发现 Java 的特别之处。你可能会预感到 Java 一定会火。投资越早，你就

越有可能成为这个新技术潮流的领导者。

这样，你的决定就是正确的。如果你做事用心，恰到好处，那你在 Java 上的投资会给你带来可观的收益，也就是我们所说的高风险，高回报。

还是 15 年前，假如你看到了 Be 公司新产品 BeOS 的演示，那个时候这是个令人赞叹的产品。利用多处理器技术，这项产品强大的多媒体处理能力令人震惊。这个平台一鸣惊人，评论员们也开始头晕目眩，预测这项技术必将成为操作系统中的有力竞争者。有了这个新的平台，新的编程方法、新的 API 和新的用户界面概念也就应运而生了。要学的东西很多，但是看起来这些努力似乎都是值得的。你倾注了大量的努力来成为第一个创造 FTP 客户端，或者是第一个创造 BeOS 个人信息管理系统的人。当 Be 公司刚发行了与 Intel 兼容的操作系统时，就开始有传言说 Apple 要收购这家公司，使用它的技术作为新一代 Macintosh 操作系统的基础。

但结果是 Apple 并没有收购 Be 公司。事实上，Be 公司的产品就连高度专门化的小市场也没能打进去。这个产品没有得到进一步发展。那些为 BeOS 环境编程的开发人员慢慢痛苦地认识到，从长远看，他们的投资不会得到回报。最后，Be 公司被 Palm 收购，这个操作系统也无疾而终。BeOS 是一项高风险但是极具吸引力的技术投资，但是对那些投资者来说，这项新技术并没有给他们带来具体的长远收益。这就是高风险，零收益。

现在，我已经谈论了选择一项全新但是不稳定的技术和选择稳定成熟的技术的不同之处。选择一项已经进入商业生产流程的稳定技术，投资风险很低，但是与投资那些无人开发的很炫的新技术相比，收益也会比较低。那么，那些即将完成使命的技术呢？只需轻轻一推，这些技术就跌进了坟墓。

那谁又是推动者呢？你或许会想到最后仅剩的几位 RPG 程序员，他们都已头发花白，数着日子等着退休。而新一代的程序员可能听都没听说过 RPG，他们学的都是 Java 和.Net。不难想象，一项陈旧的即将被淘汰的技术，

它仅存的几名拥趸的职业生涯走向结束的过程，和这项技术本身走向终结的道路是一样的。

但是，旧的系统不是灭亡，而是被取代。在新旧交替的过程中，旧的系统需要与新系统对话。必须有人知道如何将新系统与旧系统融合，反之亦然。但是一般来说，新一代的程序员和那些即将退休的老程序员都不知道或者很想知道如何才能将两代系统的特点很好地融合起来。

> 无论做出哪种选择，最终目的是产生利润。
>
> *Both ends of the technology adoption curve might prove to be lucrative.*

所以，这就需要精明的技术人员来充当“技术收容所”的角色——帮助旧系统舒服且有尊严地消失。这项工作的重要性是绝对不能被低估的。就像大多数人在沉船之前会跳海一样，那些老的程序员要么就干脆退休，要么就向另一技术领域跨一步。作为一项仍然重要的技术的最后支持者，你当然是权威。但这也是极具风险的，一旦这个技术彻底退出游戏，那你就成了一种根本不存在的技术的专家了。但是，如果你行动得够快，还可以选择下一个正在衰退的系统，然后再来一遍。

选择是把双刃剑，决定权还是在你手里。

练 习

基于当今市场，按照从左往右的顺序尽可能多地列举出处于早期、中期和晚期的技术。最左边为崭新的尚未稳定的技术，最右边为即将退出市场的技术。尽可能仔细地找到它们之间的细微关联。

当你列举出所有你能想到的技术后，标记出你认为自己擅长的技术，然后换一种颜色，标记出那些你做过但是并不精通的技术。你的标记主要集中在哪个区域？它们是聚集，还是分散的？处于这张图表边缘处的技术，有没有你感兴趣的？

2 供应和需求

Web 被广泛使用后，你只需为公司创造一个简单的 HTML 就能挣不少钱。每个公司都想拥有自己的网站，但很少人知道怎么制作。各家公司都愿意高薪聘请有经验的网页设计师。那时候只需知道基本的 HTML、超链接和站点结构，就可以称为有经验的 Web 设计师了。

制作 HTML 非常简单。制作出好的网页不容易，但是基础的东西很好掌握。那时候，Web 设计师供不应求，工资极具诱惑力，越来越多的人开始阅读相关书籍自学 HTML。结果，越来越多的人成为 HTML 方面的专家。

当 Web 设计师越来越多时，就开始划分真正具有艺术性的设计师和实用主义设计师。竞争也降低了他们的薪酬。由于雇佣 Web 设计师价格低廉，越来越多的公司开始要创建自己的网站。以前他们或许要付 5000 美金才能制作他们的第一个网站，现在只需付 500 美金。

当然，也有公司仍然愿意花大价钱制作出色的网站。那些优秀的设计师也有资本开出高价钱。

最终，网页设计师的薪酬降到了中低水平。一般水平的 Web 设计师逐渐被最终用户以及做 IT 但并非专业做网页的人取代。这样，HTML 设计者的供应、需求和价格达到了平衡。

Web 设计师行业的历史发展证明了一个众所周知的经济规律——供求规律。提到供求，大都数人都会想到一件商品的价值是多少，应该卖多少钱。如果市场上这种商品供大于求，价格就会下降；如果供小于求，那么价格就会上涨。

除了可以预测商品和服务的价格，供求关系的规律还可以预测价格的变化将如何影响出售和购买此种商品或服务的人数。通常，同一件商品的价格越低，购买者越多。

不需在价格上竞争，你承受不起。

You can't compete on price. In fact, you can't afford to compete on price.

这条规律有什么价值呢？我们可以把编程工作外包给国外团队，将大量的廉价 IT 工作人员注入到我们的市场经济中。在国内，我们担心失去工作，但是廉价的劳动力事实上也增加了市场对 IT 人员的总体需求。同时，随着需求的增加，价格也在降低。高需求产品和服务的竞争是以价格为导向的。在买方市场，价格就是薪水。你不能在价格上与他们竞争，因为你承受不起，那怎么办呢？

国外市场为我们的市场注入了廉价的开发人员，但是涉及的技术范围很窄。印度有很多的 Java 和.NET 程序员，也有很多 Oracle DBA。在国外从事非主流技术的人员还是很少的。当选择专注于哪种技术的时候，你要仔细考虑供给增长和价格下降给你的职业前景带来的影响。

作为.NET 程序员，你会发现自己每天都在和成千上万的人竞争。但如果你是 Python 程序员，那么竞争就小得多。这会造成.NET 程序员的平均工资大幅降低，也就可能会引起市场需求的增长，也就是说，会产生更多的.NET 工作机会。这样，你可以很快地找到一份工作，但是薪水不会令人满意。相对于市场需求来说，Python 程序员的供给比.NET 少得多。

如果 Python 工作能提供更高的薪水，那么就会有更多的人为了追求更高的薪水来做这份工作，这样就加剧了竞争，也会降低 Python 程序员的薪水。

这就是供求平衡。但到目前为止，印度专门为已经平衡的 IT 市场提供

服务。在印度，主流的外包公司不会着手做新技术。他们从来都不做第一个吃螃蟹的人。他们等待技术服务市场平衡，然后再用极其廉价的编程成本打入这个市场。

这样说来，你可能会选择市场上需求较低的工作。如果你害怕失去工作，自然而然地，你就会选择避免与外包公司做相同的工作。既然外包公司的工作都是市场上需求较高的，那么你就应该关注那些特殊领域的技术。这样或许不能减轻竞争压力，但是竞争的重点会由价格转向能力——这正是你需要的。你无法在价格上与他们竞争，但是可以在能力上与之抗衡。

同样地，随着主流程序员平均成本的降低，需求就会增加。对 Java 程序员整体需求的增加，会导致国内工作机会的增加。国外廉价 Java 程序员的增加可以拉动市场需求，包括对更高级程序员的需求。

现实正是如此。许多公司看到要使国外团队更好地工作，它们就必须要留住国内那些更高级的程序员。这些高级程序员可以制定标准、保证质量、领导技术团队。市场对 Java 程序员整体需求的增长，会导致对此类高级开发人员需求的增长。低端工作可能会流向国外，但比起外包之前，市场上会多出更多的高端工作机会。与特殊技术市场的情况类似，从事高端层面的 Java 开发工作，竞争就会从价格转到能力上。

发现市场上的不平衡。
Exploit market imbalances.

从供求规律中，我们可以学到重要的一点——需求的增长会加剧价格的竞争。如果只想做稳定可靠的工作，并且跟随着工作发展，那么你就会卷入与国外开发人员的价格竞争中，因为你的技术决定了你只能进入平衡的外包市场。如果在主流技术市场中竞争，你就必须在更高层面上竞争，否则，你就要去发现市场上的不平衡，找到外包公司无能为力的工作。这两种情况，你都必须找到工作的动力，提高自身的技术和灵敏度来应对一切变化。

练 习

研究当今技术市场的需求。利用招聘广告和招聘网站找出哪些工作是高需求，哪些是低需求的。登陆外包公司的网站（如果你在这些公司工作，可以直接与员工交流），把这些公司的技术与你发现的高需求工作进行比较。记录下那些在国内市场中高需求且没有流到外包市场的技术。然后再将这些外包公司的技术与前沿科技相比较。密切关注外包公司还没有涉足的上述两类技术。思考它们需要多长时间才能为相应的市场提供服务。这个时间差就是市场不平衡的阶段。

3 只会编程是不够的

只思考在哪种技术上投资是不够的。毕竟，技术只是一种商品。你不可能只掌握一种编程语言，或者只能够操作某种系统，然后把生意交给老板打理。如果他们只想找个懂代码的机器人，那不如雇个外国廉价的程序员。如果你想站稳脚跟，必须要深入了解你所处的领域。

事实上，软件工程师不能只会开发软件，应该要成为这个业务领域的专家。在我之前工作过的一家公司里，就有这么一个团队。我第一次见到这个公司数据库管理团队的时候有点儿震惊，因为这个团队里的成员都相当厌烦数据库技术。我当时在想，既然是这样，那这些人为什么要干 IT 呢？单在技术上讲，他们算不上出色，但是这个团队有他们的特别之处。作为企业数据的保存和维护人员，他们比那些商业分析师更加了解这个行业。他们的知识和对这个行业的了解使他们成为了数据管理工作的抢手人才。我们这些愚人居然还看不起人家。他们做的工作正是他们的价值所在。

你的行业经历应该成为你的重要才能。如果你是搞音乐的，当你描述你的才能时，不能只说我能演奏某首曲子，而要说你真正了解这首曲子的内涵。商业领域的经验也是一样。比方说，如果你正在做一个医疗保健项目，你能区分出 HIPAA835 和 HIPAA837 这两种电子数据交换（EDI）协议有什么不同吗？同是软件开发人员，这个知识不就能决定谁更适合这个职位了吗。

或许你只是一个程序员，但是如果你能用客户所处行业的专业语言与他们交流，那这就是一项非常重要的技能。就像如果与你工作的人都真正了解软件开发是怎么回事，你会不会觉得一切都会变得更加得心应手呢？你再也不用向他们解释为什么不能在 Web 应用程序中的一个页面上返回 30 000 条记录，或者解释为什么他们不能向开发服务器发送链接。你的客户也有同感。

换位思考一下，如果你是客户，为你服务的程序员了解你的行业，不用什么都得由你来做决定，你也不用紧张担心哪个小细节会出问题，你会不会觉得工作起来更容易呢？你从事的行业也是这样。

现在的软件开发界，Java 和.NET 大行其道。如果你会这两门技术，那你就能在使用这两项技术的公司找到工作。选择商业领域也是同样的道理。在选择从事哪个行业的时候，你应该像选择掌握哪门技术时一样谨慎。

鉴于行业选择是十分重要的，那么选择在哪个公司、哪个领域工作对你来说也是重要的。如果你还没有仔细考虑过这个问题，那现在开始思考吧。机遇每天都在流逝。就像利息马上就涨了，但你却把钱存在了一个低利率的死期账户里。把自身的发展限制在一个静止不前的行业里，可不是什么好的投资选择。

> 仔细思考在哪个商业领域投入时间。
>
> *Now is the time to think about business domains you invest your time in.*

练　习

(1) 安排一次与业内人士的午餐，问问他们是如何工作的。交流中，思考如果你来做他们的工作，你会做什么改变或者你可以从他们身上学到什么。询问他们日常工作中的细节。问问他们技术是如何帮助（或者阻碍）他们工作的。从他们的角度出发，思考你的工作。

定期安排此类活动。刚开始你可能会觉得有些尴尬，但没关系。我是几年前开始这么做的，这极大地帮助我理解和融入我所服务的行业。另外，在与我的客户交谈时，我也变得更加得心应手。

(2) 选择一本与你公司行业有关的杂志。你甚至都不用买，大多数公司都有些过期的行业杂志。试着阅读它们，虽然有些东西你可能不懂，但是要坚持。列出你可以向客户询问的问题。不要担心你的问题很傻，客户会大为

赞赏你的这种学习态度。

找一个你可以随时登录的行业网站。无论是浏览网站时，还是阅读杂志时，注意大事件和专题文章。你所处的行业正在为什么而努力？现在的热门是什么？不管是什么，把它们介绍给你的客户。请他们说说观点看法。思考这些潮流是如何影响你的公司、你的部门、你的团队，以及你自己的工作的。

4 做团队中最差的

爵士乐的传奇人物，爵士乐吉他手 Pat Metheny 给年轻音乐演奏者提出了一条建议——“做乐队中最差的乐手。”[①]

> 做乐队中最差的乐手。
> Be the worst guy in every band you're in.

进入 IT 这行之前，我是一名专业爵士和蓝调布鲁斯萨克斯风演奏者。作为一名乐器演奏者，我很幸运地早早就学到了这个道理并且一直坚持这么做。做乐队中最差的乐手意味着你总是在与比你优秀的人一起演奏。

这样的话，你为什么不选择做这个最差的乐手呢？你会问“那这样不就会承受很大的压力么？”没错，刚开始压力是很大。作为一名年轻的乐手，我总是十分显眼，因为我总是乐队中最差的乐手。去演出时我连萨克斯风都不想拿出来，因为我怕被人赶下舞台。那个时候我总是仰视身边的人，期望有一天自己也能达到他们的水平，甚至梦想能成为乐队主奏。

感谢上帝，我没有失败。神奇的事情发生了。我在这行占有了一席之地。我不是乐队中最差的那个，但也没有成为最优秀的。这有两个原因，其中一个原因是我并不是自己想象得那么差。这点我们稍后再讨论。

更有意思的原因是我的演奏可以自动模仿我的偶像演奏出来的音乐，这使我在他们中间占有了一席之地。我希望这是因为我自身具有某种超能力——站在一个天才旁边，就能拥有他的能力。但回想起来也没这么神奇，这好像就是出于一种本能。就好像如果我周围的人说话方式与我不一样，那我就会自然而然地受他们影响，说话时使用他们的词汇或者语法习惯。我曾

① 原文见 Chris Morris 的博客 http://clabs.org/blogki.

在印度生活过一年半，从印度回来后，我妻子经常被我说的话逗得哈哈大笑，她问我："你听见自己刚才说什么了么？"我居然在讲印度英语。

我做萨克斯风手时，就是做乐队中最差的演奏者。我只能像其他人一样演奏。事实上当我在赌场或者巴掌大的酒吧里与那些差劲的乐队一起演奏时，我的演奏水平也向他们靠拢。我发现就算不是在酒吧演奏，我也摆脱不了从那些差劲的乐队那里染来的坏习惯。就好像那些酒鬼，清醒的时候说话也含糊不清。

所以我认识到人们会取得很大的进步或者退步，仅仅是因为与他们合作的人不同了。与一个团队合作的时间长了，会对自身的能力产生持久的影响。

> 你身边的人会对你产生很大的影响，明智地选择你的圈子。
>
> The people around you affect your own performance. Choose your crowd wisely.

作乐手的时候，我养成了寻找最好的乐手与之一起演奏的习惯。进入 IT 这行后，这种习惯自然而然地延续了下来。我下意识地去寻找最棒的 IT 人士，并与他们一起工作。显然，真理是禁得起考验的。做编程团队里最差的程序员和做乐队里最差的乐手产生的效果是一样的。你会发现自己变得出奇地睿智。你写的东西，和你的谈吐都会变得越来越有智慧。你编写的程序和设计会越来越高雅优美。你会越来越有创造力，难题也迎刃而解。

好，现在我们回到能让我意想不到地融入乐队的第一个原因。我确实不像自己想象的那么差劲。在音乐这行，要想得到别的乐手对你的真实评价，并不是件难事。你优秀，那人家就会再次邀请你合作；你差劲，别人就会避免和你合作。比起你直接问起他们如何评价你，这种检验方法更能得到真实的反馈，因为好的乐手不愿意和差劲的乐手同台。让我吃惊的是，很多优秀的乐手都会再次邀请我和他们同台，甚至邀请我与他们一起组建乐队。

试图做一个团队里最差的人可以让你不再小看自己。可能你的能力应该是在甲等乐团演出，但你自己却认为自己属于乙等乐团，这都是因为你恐惧。清楚地知道自己不是最好的，就不会总担心被人发现你不是那么优秀。事实上，即使你在尝试做那个最差的，也并不意味着你就是最差的。

练 习

找一个团队，让自己成为“最差”的。不需要立刻调换工作，你可以试着找一个志愿者项目，通过与这个项目中其他程序员的合作，提高自身能力。查查有哪些编程团队会议，然后去参加这些会议。程序员一般都会用业余时间做兼职，以此来练习新的技术，提高自身技能。

如果在身边找不到这样的程序员组织，就利用网络。找一个你钦佩的开源项目，且他的设计者是你下一阶段发展的目标。浏览这个项目的待处理列表和官方讨论区，或者编写一个功能或者修正一个大的错误。你的代码要模仿这个项目的代码风格，但是又要让你的代码和设计与其他项目完全不同，甚至让原作的程序员都认不出来。在你觉得一切都妥当之后，把它作为一个补丁提交。如果你做得好，这个项目就会接受它。这样重复来做。如果这个项目的设计团队不同意你的观点，那就将他们的反馈加入到你的设计中再次提交，或者记录下他们做出的改变。最终，你会发现自己成为了这个项目团队中值得信赖的一员。你会惊喜地发现虽然这些高级程序员并不在你的身边，你甚至连他们的声音都没听过，但你已经从他们身上学到了很多东西。

5 在思维上投资

当你选择专注于哪个领域发展的时候，那些容易找到工作的技术很吸引眼球。Java 和.NET 都是很强大的。学习 Java，你就可以去申请一份编写 Java 代码的工作，而且成功得到这份工作的几率很高。

这样想，那如果在一种新的还未稳定的技术上花费时间和精力，就会显得很愚蠢，特别是如果你之前并没打算开发这种技术。

TIOBE Software 利用网络搜索引擎，根据全球范围内有经验的工程师、课程和第三方供应商对程序设计语言的实际使用率，将编程语言做出排序[①]。这种统计方式虽然不是很科学，但可以起到很好的指示作用。

当我还在撰写此书的时候，最受欢迎的编程语言是 Java，C 语言紧随其后，C#位列第 6 位，但是已出现微小的上升趋势。SAP 的 ABAP 位列第 7 位，但名次成缓慢下降趋势。我最喜欢的 Ruby 排名第 11 位。在重要的工作中我一般都用 Ruby，并用它来做每年国际性会议的议题。但当本书第一版发行时，Ruby 居然跌出了前 20 名，位于 ABAP 之后！

这么说来，我使用 Ruby 只能说明我疯了或者傻了？Paul Graham 在 *Great Hackers*[②]一文中曾宣称使用 Java 的程序员没有使用 Python 的程序员聪明，这一观点在这个行业中引起了一阵骚动。他惹怒了很多愚蠢（不敢相信我自己居然这么说）的 Java 程序员，他们在自己的网站上驳斥这一观点。这种反击行为恰恰证明 Paul Graham 触动了这个行业的一个敏感点。当他第一次以演讲的方式发表这篇文章时，我在现场，他让我回想起了往事。

---

① http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html。

② http://paulgraham.com/gh.html。

一次我去印度招聘，要从上百名面试者里挑选十几名适合的人选。整个招聘团队筋疲力尽，因为大家费尽周折，却根本没有挑选到适合的人。我们头疼欲裂，眼睛也熬得通红，晚上开会商讨应该如何改变面试的策略。为了面试更多、更优秀的候选者，我们需要优化面试流程。我连续 12 个小时努力让那些紧张沉闷的应聘者开口说话，嗓子都哑了。所以我提议在猎头简历搜索库的关键词中增添“Smalltalk”，但是人力资源总监的答案是“在印度，没人知道什么是 Smalltalk。”这就是关键所在了。没人知道 Smalltalk，用 Smalltalk 编程与用 Java 是完全不同的。这种不同的经验使我们对候选人的期待值不一样，Smalltalk 环境的动态特征赋予 Java 程序员在处理问题时一种新的思维的方式。我希望这些因素能够让我发现技术成熟的程序员，但在这之前我还没找到符合条件的应聘者。

在搜索关键词里增加了“Smalltalk”后，大大缩小了候选范围。符合条件的应聘者真正理解什么是面向对象的程序设计。他们认识到 Java 不是能解决任何问题的万应灵药。他们中的大多数人真正热爱编程！招聘团队就像发现了未经打磨的钻石，心里想，前两个星期你们都干嘛去了！

由于他们很优秀，所以有资格提出条件。可惜，我们给出的薪酬有限，不足以吸引他们。大都数人都选择留在原来的公司或者继续寻找工作。尽管没能留住他们，但我们学到了宝贵的招聘经验：比起那些经验单一的候选人，我们更倾向于那些具有丰富经验的候选人。我认为优秀的程序员之所以寻找变化和多样性的工作，是因为他们喜欢学习新东西，或者是因为他们很清楚要想成为更加成熟、更加全面的程序员，就必须去学习新的技术、在新的环境下工作，获取新的经验。我认为这两方面因素都奏效。现在我仍然使用这个技巧来招聘程序员。

所以你与其千方百计地想要进入我的候选人范围，不如把精力放在学习以前没有使用过的技术上。

作为招聘经理，我认为判断你适合不适合一个职位的首要因素就是你是否对这行感兴趣。如果我知道你为了自身发展，或者更理想的是，你单纯因为兴趣而学习新的东西，我就会知道你热爱你的职业，把你的职业视为动力。当我问候选人有没有用过某种非主流的技术时，最不愿意听到的答案就是“没有人给我机会使用”。没有机会？从来也没有人主动给我提供过这种机会啊！机会是要自己争取的。

> 没人给过我机会……？要学会抓住机遇！
>
> I haven't been given the opportunity...? Seize the opportunity!

除了可以激励你，使你更加热爱工作，更重要的是，接触这些边缘技术和方法能让你更有深度、更加优秀、更具智慧，以及更具创造力。

如果你认为这样还不足以成为你学习新技术的原因，那或许你选错了职业。

练 习

学习一种新的编程语言。但不是从 Java 到 C#或者是从 C 到 C++。这门新的语言应该可以让你的思维方式产生变化。如果你是 Java 或者是 C#的程序员，那就尝试学习类似 Smalltalk 或者 Ruby 这种不需要采用强类型的静态编程方式的语言。或者，如果你一直在做面向对象开发的话，可以尝试 Haskell 或者 Scheme 这样的函数式语言。你不需要成为专家，可以感到这种新的编程环境与你之前所处的环境的不同之处即可。如果你觉得并没有什么不同，那就说明你选错了语言或者你仍然将固有的思维方式运用到新的语言中。要彻底改变你的思维方式来学习新的语言。向熟悉这些语言的程序员请教，让他们检查你的代码并提出建议，使之更符合此种语言的特性。

6 不要听从父母

我们的文化要求我们听从父母的建议。这就是一个孩子的职责——做应该做的事情——就像是一种虔诚的信仰。图书、电影和电视情节都在讲述或上演着父辈的智慧。但在我们这个行业，这条真理是行不通的。

父母总不希望儿女去冒险，所以他们并不期望儿女有一个多么卓越的职业，只要差不多就行了。比起其他人的建议，父母给的建议总是包含着种种担心。这种出于担心的建议目的就是不要让你经历失败。但想着如何避免失败绝对不是取得成功的方法！成功是要冒险的。胜利者想的是他们想要做什么，而不是其他人会怎么做。出于担心的职业规划不会让你走向成功，而是会局限你的发展。没错，这条路很安全，但毫无乐趣而言。

在上一代人选择职业的时候，乐趣绝对不是一个决定因素。工作不是用来产生乐趣的，而是为了填饱肚子。工作之余才会谈到乐趣，那是晚上下班后和周末的事情。但是后来我们认识到，如果工作没有乐趣，那我们就没有动力去做好它。现在，这种观念虽然没有什么大的改变，但是我们的文化在如何看待工作的意义这个问题上向好的方向转变了。越来越多的人懂得了只有对工作充满激情，才会做出卓越的工作。在软件这行，如果没有乐趣，那工作起来就不可能充满激情。

另外一个决定职业规划的因素是跳槽，父母也不会赞成这个观点。一个成熟的职业软件开发人员需要从各个角度了解这个行业：产品开发、IT 支持、内部业务系统开发以及管理工作。作为软件开发人员，你看到的角度越多，攻克的技术难题越多，就意味着你越有足够的能力来面对艰难项目。对一个程序员来说，只在一个公司工作，加强单一业务技能，会局限职业发展。“铁饭碗”的时代已经不复存在了。以前，这种进入某个公司并为其服务终身的行为被看作是一种奉献。但现在这是一种障碍。如果你只在一个公司工作过，只看到了一种系统，那么当那些明智的经理决定是否要雇用你的时候，它就

成为了一个不利因素。就我个人而言，比起那些只知道一种做事方法的人来说，我更愿意聘请在不同的环境中经历过成功与失败的人。

几年前，我开始认识到我父母那一代人的职业价值观很大地影响了我的职业规划。那时候我受雇于全球规模最大且稳定的公司之一，并且慢慢稳步地升迁。但我发现我好像没有什么前途。我安慰自己说这个公司这么大，我的发展不会受到局限。我可以在不同的工作地点做不同的工作，但是最终，我还是在同一个地方做着相同的工作。

我曾经和一个朋友提起过想换个公司，他却说："没准你下半辈子注定要在大公司工作。"上帝，不要啊！于是我马上找到了一份新的工作，离开了原来的公司。

这也是我在软件界走向成功的标志性起点。我看到了以前从来没有见过的领域，我开始应对更难的问题，我得到的回报比以往任何时候都多。刚开始我确实有点害怕，但是当我改变了出于担心的保守职业规划后，我的事业和我的生活都变得更加美好了。

在职业道路上，需要一些有目的性的冒险。别让恐惧征服了你。如果在工作中没有感到乐趣，那你就不可能出色地工作。

练习

在职业道路上，你最担心什么？回想你最近做过的几次职业选择，不用是很大的决定（如果你是出于某种担心而做出职业的选择，那也不会是什么大决定）。你从事了什么特殊的工作，或者你申请了一份新的工作或升职。把这些选择罗列出来，逐一做出诚实的评价：这些决定受到"担心"这一因素的影响有多少？如果你没有担心，那你会做到什么程度？如果这些决定确实是受到"担心"的影响了，那你现在如何逆转它，寻找新的机会做出新的选择，当然这次不要再因为担心什么而受到约束。

在微软 30 万美金的诱惑前，我却选择了 GitHub

——Tom Preston-Werner，GitHub 的创始人之一

2008 年是闰年，也就是说在 366 天前，几乎不差一分钟，我一个人坐在旧金山第三大街的 Zeke's 运动酒吧的包间里。我不经常去运动酒吧，但那个周四是“我可以谈论 Ruby”之夜。我当时想“我可以……”总是个好事。ICHR 是个半私人的会议，来的人都是志同道合的 Ruby 爱好者，这个聚会一般都会延续到晚上，大家通宵达旦地饮酒。通常这种晚会会如同我第二天早上的宿醉一样烟消云散，但是这个夜晚不同，因为它是 GitHub 诞生之夜。

一开始我说自己一个人坐在包间里是因为那天我们的长桌在酒吧靠后的地方，灯光昏暗，我刚点了杯 Fat Tire，需要跳出那个社交氛围休息一下，所以感觉就像是一个人坐在包间里。我小饮了几口，Chris Wanstrath 走了进来。我现在想不起来那时候我和 Chris 是不是已经成为了朋友。我和他在关于 Ruby 的聚会和会议上见过面，但只是泛泛之交。我也不记得当时为什么就朝他做了个手势让他过来，可能是因为我觉得他编的程序很不错。我对他说：“老兄，看看这个。”大约一星期前，我刚开始做一个叫做 Grit 的项目。这个项目使我可以通过 Ruby 代码以一种面向对象的方式访问 Git 的代码库。Chris 是当时很少的几个开始认真关注 Git 的 Ruby 专家。他坐下来，我开始把我所做的工作展示给他。我做的东西不多，但显然，它已经足以激起 Chris 的兴趣了。看出这点后，我开始向他讲述一个不成熟的设想：我想做一个类似网站的东西，这个网站作为中心，程序员可以在上面分享他们的 Git 代码库。我还给它起了个名字——GitHub。当时可能我还在解释什么 Chris 就打断我说：“算我一份，就这么干！”而且说得异常坚定。

第二天——2007年10月19日星期五，晚上10点24分——Chris带来了他的第一笔投资，用数字石雕为我们的合伙事业盖上了印章。到那时为止，我们两个人还只是想在一起编程，没讨论过这个计划要如何进行。

还记得电影《小子难缠》（The Karate kid）中，丹尼被训练成武术大师的情节么？那几分钟非常精彩。记得那段音乐么？或者你应该去听听Joe Esposito的You're the best，因为接下来的故事非常蒙太奇。

接下来的三个月里，我和Chris花了大把的时间在GitHub的设计和编程上。我继续研究 Grit，设计出了用户界面。Chris 构建了Rails 应用。我们每周六见面讨论设计方案，试图给这个项目做出预算。一个大雨天，我们俩吃着美味的越南蛋卷，就定价策略这个问题谈论了两个多小时。那时候，我们俩还都有各自的正式工作。我在 Powerset 公司为 Ranking 和 Relevance 团队开发工具。

3个月没日没夜的工作之后，到2008年1月中旬，我们推出了一个私人试用模式，并向我们的朋友发出了邀请信。2 月中旬，P.J. Hyett 加入了我们，壮大了我们的团队。4 月 10 日我们正式启动了这个网站，但没有通知 TechCrunch 网站。这时候，这个项目还只是由三个20多岁的小伙子创立的事业，没有半毛钱的外界投资。

2008年7月1日，我还在Powerset做着全职工作，那天微软公司花费大约1亿美金收购了 Powerset。这就有趣了。我之前就知道早晚得做出去留的抉择，但没想到会这么快。我要么与微软签合同成为他们的一名员工，要么就辞职，全职做 GitHub。那年我29

岁，是3个人中最大的，也是3人中欠债份额最多、月消费最多的一个。我已经习惯了年薪 6 位数的生活方式了。另外，我妻子在哥斯达黎加的考察工作马上就要结束了。我很快就要从一个貌似单身汉的身份回到已婚男人的生活。更让我难以做出决定的是，微软开出的薪酬十分诱人——薪水再加上工作超过 3 年奖励 30 万美金。这么具有诱惑力的条件我相信任何人都会三思而后行。所以当时我面临的选择是：一份在微软稳定且高薪的工作，或者不知道要投入多少资金并且极具风险的事业。另外两位伙伴在有了些积蓄后，他们就不再做全职工作了，全身心地投入于 GitHub。所以我知道我待在 Powerset 的时间越长，对另外两个伙伴来说压力就越大。这是决定“做或者放弃”的关键时刻。选择 GitHub 并为之奋斗，或者做个安全的选择——在微软工作，让荷包满满，放弃 GitHub.

如果你想要个睡不好觉的秘方，我这有：把“我妻子会怎么看待这个问题”加上 3000 张百元美钞中，再加上你最爱的啤酒，最后放入一个清偿债务的诱惑。

如今，对于如何向老板提出辞职，我已经得心应手了。那天当我接到继续受雇的通知时，我告诉老板我要辞职，全力去搞 GitHub。我的老板很好，他虽然有些吃惊但还是对我表示理解。他没有用更高的奖金来诱惑我。我觉得他心里知道我是非走不可了。因为我想要离开，所以比起留在这里，新机会给我的诱惑更大。微软的经理们是非常狡诈的。他们对如何给出保留奖金相当在行——因为我想要离开，所以比起其他人，新老板给我的诱惑可能更大。告诉你，微软的经理非常精明。他们对如何提供保留奖金非常在行。当身边有这么一个人的时候，事情就会变得有些古怪。

最后，就像印第安纳琼斯永远不会放弃寻找圣杯的机会一样，就算另一个选择再稳妥，对于我真正热爱的事业，我也绝不会放弃。等我老了，驾鹤西游之前，回想过去我希望我会说“上帝，这辈子真是险象环生啊！”而不是“嗯，这辈子过得还算稳稳当当。”

7 做一名通才

至少 20 年前，绝望的经理和老板一直欺骗自己说软件开发是一种机械化的生产流程。制定出规格标准，架构师把这些规格标准转化为高层次的技术层面。设计师再把详细的设计文档填入这个框架，然后交给像机器人一样工作的程序员，他们一只手拿着低俗小说，另一只手慵懒地敲入设计执行方案。最后，Inspector 12 收到完整的编码，经测试符合最初的规格标准后，准许通过。

经理们都希望软件开发机械化，这并不是什么奇怪的事。他们了解怎么做好机械化工作。几十年的经验教会了我们如何有效精确地制造有形的东西。所以，把我们从机械生产中学到的经验运用到软件开发上，我们就可以将其优化成一系列的生产流程。

在这个所谓的软件工厂中，雇员都是专才。他们坐在流水线旁自己的座位上，把 Java 的部件组合在一起，或者在软件车床上打磨一个 VB 的应用程序。Inspector 12 是测试员。软件组件随着流水线向下流动，Inspector 12 每天以同样的方式测试这些组件并盖章以示合格。J2EE 设计师设计 J2EE 的应用，C++的编码师在 C++环境下编码。这个工厂中，一切都划分得很分明，安排得有条有序。

但是，这个类比并不成立。软件至少应该适应软件需求。这个行业已经变了，商业人士知道软件很“温柔”，可以根据需求做出改变。但这也就意味着构建、设计、编码和测试环节也要相应地变得更加灵活，这些都是机械化生产过程无法满足的。

在变化如此迅速的环境下，灵活才能制胜。聪明的生意人在碰到难题时，会向身边的专业软件师寻求帮助。那么，你怎么才能成为这些生意人遇到困

难时首先想到的“英雄”呢？答案就是——能够解决一切可能出现的难题。

但是这些难题是什么呢？没错，你我都一样，我们无法预测会产生什么难题。我只知道这些问题非常多样化，诸如严重的设计缺陷需要立刻修补，异构系统的集成，以及 ad hoc 报告的生成。面对这么多种问题，可怜的 Inspector12 可能会命不久矣了。

有句话是“什么都懂点，但什么都不专”，一般来说，这句话是贬义的，是说这个人没有专注于某一项领域，并深入学习，成为这方面的专家。但是，当你的购物网站“提交订单”出了故障，每个小时你都会损失上百个订单时，那这个“什么都懂点，但什么都不专”的人可能既知道这个程序代码是怎么运行的，还会做些简单的 UNIX 调试，分析 RDBMS 规范中潜在的性能瓶颈，并能检查网络路由器配置看是否存在某些隐蔽的问题，更重要的是，找出这些问题后，这个人可以很快做出架构和设计决定，纠正代码，部署一个新的系统。这样看来，机械化生产模式看起来就非常奇怪，而且具有很多的缺陷。

另外一个机械化生产模式无法立足的原因是：这种生产线使工作按照稳定的步伐直线进行，而软件项目通常是具有循环性的。不仅项目的流动是循环的，一个项目内部的工作也是循环的。在制定完软件的规格、架构和设计之前，程序员要么坐在椅子上等，要么在这段时间着手做其他项目。但这种同时参加多个项目的问题是，不管这个软件的开发目的是什么，当程序员要大展身手的时候，必须要依赖前后流程和经验。规格、架构和设计文档可能非常出色，但是如果程序员不懂这个系统是用来做什么的，他就不能很好地实现这个系统。

当然，我所说的不仅适用于程序员。软件开发上的任何一个职位都是如此。由于前后流程的问题，同时参加多个项目并不可行。结果是，我们的生产系统是低效率的。在机械化生产模式中，有各种各样的方法尝试解决效率低下的问题。但是，我们还没有想出办法来优化我们的软件工厂，使它变得

更有效率。

如果你只是一名程序员、测试员、设计师或者架构师，那你很可能会坐在那里无所事事，或者当你的项目快要结束时，你却在忙忙碌碌。如果你只是一个 J2EE 程序员或者是一个.NET 程序员，或者是 UNIX 系统管理员，那当一个项目或者一个公司的关注点开始转移出你擅长的技术领域时，你就会发现你不再发挥作用了。这不是说在一个项目的流程中，你的价值有多大(架构师的价值往往最大)，而是说你可以在多广的范围内发挥作用。

如果你想在这个行业站稳脚跟，那我建议你要成为通才。如果你害怕你的部门裁员，那你就该知道精简团队的时候，一个只会测试或者只会编码的人肯定会被裁掉的。如果你就是单纯地想要卓越，那更好，你要动动脑筋掌握大局。

> 通才很少，所以很珍贵。
> Generalists are rare...and, therefore, precious.

成为通才就是说让你不要只专注于一种技术。在工作中，有很多方法可以让我们扮演多种角色。为了使成为通才这个概念形象化，我们可以把 IT 职业分解成几个独立的部分。我想到了五个，但肯定还有更多，就看你是如何划分了：

- 职业阶梯的各层
- 平台和操作系统
- 代码和数据
- 系统和应用
- 业务和 IT

这些不同的方面可以帮助你了解如何成为一名通才。这只是审视职业的其中一种分类方法，你可以针对自己的情况，找到更好的方法。这里我们只对这个分类进行讨论。

首先，你可以选择成为一名团队负责人、经理、技术人员，或者一名架构师、程序员、测试员。很多人都不明白能够适应和胜任不同角色的价值所在。例如，一名强大的团队领导者应该尽力成为多面手。现在国内的编程团队十分精简，团队领导应该既能领导团队做项目，又能在外包团队偷懒的时候，卷起袖子亲自修复紧急严重的漏洞。软件架构师也一样，他要是再能写一些代码，那可能会大幅度地提高整个项目的进程。当工作要跨越职业阶梯的等级时，人们不是不愿意去做，而是没有能力去做。程序员不会领导团队，团队领导人不懂编程。能够把两样做得都很好的人，太稀有了。

你的技术水平应该超越技术平台。

Your skills should transcend technology platforms.

下一个要说的是平台和操作系统。现在如果一个做UNIX的人拒绝做Windows，那就太不实际了。同样，做.NET的也不可能不做J2EE，任何基础平台都是这样。要想在这行站稳脚，就必须做个多面手。任何人都有自己喜欢的技术，但是我们不能太理想化，自己喜欢什么就做什么并不实际。现状是我们要成为某一项技术的专家，同时还应该再擅长几种别的技术。技术平台只是一种工具，你的技术必须要高于它。如果我们想雇一个只做 Windows 的人，那我们会去国外找。如果我们想找个真正了解 Windows 和 UNIX 开发，又能帮助我们把这两者结合起来的人，我们会在国内寻找。这就是团队精神的本质。

同样，软件开发师和数据库管理员（短短 10 年间，这个职业从无发展到现在的重要地位）之间的界限也不应该划分得那么清楚。数据库管理员应该既知道如何使用 GUI 管理工具，也知道如何创建一个特定数据库产品。你不需要非常了解如何使用数据库。另一方面，软件开发师越来越忽视了解如何使用数据库了。两者相辅相成。

我刚进入这行时，首先让我感到吃惊的是，很多受过良好教育的程序员居然不知道如何安装他们用来开发和部署的系统。与我合作过的一些开发人员居然不会在PC机上安装一个操作系统，更不会安装他们用来部署应用程

序的应用服务器。一个真正懂得他工作平台的开发人员简直太少见了。要能找到这么一个人才，那做出的应用程序就会更好，工作进展也会更快。

最后，我们在本章第 3 节中提到的存在于业务和 IT 之间的那堵墙，应该立刻被推倒。现在就开始学习你的行业是如何运作的吧。

练 习

列出你能将你的知识和能力融合在一起的工作内容。写下每个方面中你的专长。例如，如果你列出了平台和操作系统，那就可以在旁边写上 Windows 和.NET。在你专长的右边，再列出你要学习的一种或几种技术，可能是 Linux 和 Java（或者是 Ruby、Perl）。

然后尽快（一周之内）找出 30 分钟开始研究你要学习的一门技术。不要只是单纯阅读相关的书籍资料，动手实践一下。如果它是种网络技术，那就下载一个 Web 服务器安装包，然后自己安装。如果是与做生意有关的话题，那就找一个你的客户，约他出来吃饭聊聊天。

8 成为一名专家

我问："仅仅使用 Java，如何编写一个程序使 Java 虚拟机崩溃？"应试者沉默了，然后无助地问："怎么可能？"

"抱歉，我没听清楚，您能再重复一遍问题吗？"这个声音听起来有些绝望。从我的经验来看，重复这个问题也起不了什么作用。但我还是提高声音，放慢语速，重复了一遍这个问题："仅仅使用 Java，如何编写一个程序使 Java 虚拟机崩溃？"

"……抱歉，我之前没这么做过。"

"我知道你没做过。那接下来这个问题呢——如何编写出一个程序，不使 Java 虚拟机崩溃？"

此次面试的目的是寻找真正优秀的 Java 程序员。面试一开始我就让这名应试者（包括我本周面试过的其他人）给自己打分，满分 10 分。他打出的分数是 9 分。我要找的是一名新星。如果一个人对自己的评价如此之高，那他干嘛不干脆编一个恶意程序，引发 Java 虚拟机崩溃？

应试者缺乏技术深度。

> **很多人认为专攻某种技术就简单地意味着不知道其他技术。**
>
> *Too many of us seem to believe that specializing in something simply means not knowing about other things.*

这就是一个声称是 Java 专家的人给出的答案。如果你在一个聚会上碰到他，问起他是做什么工作的，他会说："我是 Java 程序员。"但是，他却连这么简单的问题都答不上来，甚至连一个错误答案都给不出来。在这次紧张的全国招聘中，这种情况不是个例，而是非常普遍的。

成千的 Java 程序员申请了职位，但没有一个人知道 Java 类装载器是如何工作的，也没人能高度概况出 Java 虚拟机是如何处理内存管理的。

没错，你不需要知道这些知识，在别人的监视下编写基本的代码。但是这些人是所谓的“专业人士”。很多人都认为专于某种技术，就简单地意味着不知道其他的技术。要是这么说，那我就可以说我妈妈是一个 Windows 专家，因为她从来没使用过 Linux 或者 OS X。我还可以说我那些住在阿肯色州乡下的亲戚是乡村音乐的专家，因为他们从来都没听过别的类型的音乐。

假设你左臂下方的皮肤里长了一个奇怪的肿块，你去看家庭医生。你的医生建议你去看专科医生做个切片检查。如果这个专科医生在医学院学习的时候根本没上过课，或者他在做住院实习医生的时候没有做过你要做的这类切片检查呢？我的意思不是说他们可以从事比这个切片检查更高深的工作，要是他们只是学到了肤浅的知识，其他什么都不知道呢？你可能会问：“手术过程中，监测仪器开始哔哔作响怎么办？”这个医生的答案是：“以前没发生过这种情况，这回也不会发生的。我也不知道这仪器是干嘛用的，不过它从来没发出过哔哔声。”

谢天谢地，软件开发师做的不是性命攸关的工作。如果他们犯了错误，一般也就是导致项目超出预算，或者是造成产品缺陷使得他们的老板付出更多的金钱，而不是生命。

遗憾的是，软件开发界有很多这样肤浅的专业人士，这些人以“专业人士”为借口，只知道一门技术。在医学界，专科医生是指对某一特定领域有深刻了解的人。医生建议他们的病人向专科医生寻求帮助，因为在某些特定情况下，相比于全科医生，专科医生可以让病人得到更加专业的治疗。

那么，在软件界，什么样的人才能称得上是专业人士呢？我在招聘的时候找遍了每一个角落，寻找真正深刻了解 Java 编程和部署环境的人。我想

要寻找的人是已经处理过我们工作中可能遇到的80%的问题，并且拥有足够的知识来应付另外还未出现的20%的问题。我需要的人是不仅可以处理高水平的抽象，同时应该了解那些实现高端抽象的低端细节。我需要那些可以解决部署问题的人，或者如果他们解决不了，至少应该知道找谁来帮忙的人。

计算机界变化迅速，只有这样的专业人士才能生存下来。你是.NET 专业人士，但这绝不能成为你除.NET 之外对一切一无所知的借口，而是说，你是.NET 的权威，但当 IIS 服务器需要重启时，对你来说是小菜一碟。有人问你怎么用 Visual Studio .NET 进行源控制集成时，你的答案是："我做给你看。"由于不满意应用性能，客户提出要退出项目，这时候，你只需要三十分钟就能把问题解决。

如果你做不到以上这些，那以后请不要再顶着专业人士这个头衔。

练 习

(1) 你是否使用在虚拟机上编译并执行的编程语言？如果你使用，花点时间学习虚拟机内部是如何工作的。很多书籍和网站都专门就 Java，.NET 和 Smalltalk 进行讨论。学习这些东西总比你凭空想象要简单。

不管你使用的编程语言是不是依赖虚拟机，花点时间学习编写源文件。你敲打出来的代码是如何从可阅读的文本转变成可被计算机执行的命令的？编写你自己的编译程序又意味着什么？

当你输入或使用外部函数库时，它们是从哪里来的？输入一个外部函数库到底意味着什么？你的编译程序、操作系统或者虚拟机是如何将多个代码段连接起来，形成一个连贯系统的？

掌握这些知识可以使你在技术选择上向"专业人士"跨近一步。

(2) 在工作中或者工作外寻找一个教课的机会。你所传授的知识是自己想要深入学习的技术。在第2章第14节我们会讲到，讲课是最好的学习方法。

9 切忌孤注一掷

在我负责管理一个应用程序开发团队时，曾问过我的一名雇员："你的职业规划是什么？你将来想要成为什么样的人？"他说："我想成为一名J2EE 架构师。"这个答案让我十分失望。我问他为什么不想做一名"微软Word 设计师"或者是一名"RealPlayer 安装者"。

这个人想要把自己的职业道路建立在一门特定的技术上，这门技术是由一家特定的公司创造，而他自己又不是这家公司的雇员。这家公司要是停业了呢？如果这家公司现在热门的技术有一天过时了呢？为什么要把自己的职业发展完全依赖于一家技术公司呢？

不知道为什么，在这个行业中，我们常常欺骗自己说市场的主导和标准是一个概念。所以一些人就认为把其他公司的产品作为自己产品的一部分是合理的。更有甚者，把自己的职业发展建立在非市场领导的产品上。到事业惨败时，他们除了思考自己失败的职业规划，别无选择。

之前我们讨论过，我们应该把自己的职业规划当作是一门生意。尽管我们创立的生意可以寄生于其他生意（比如那些创造移除间谍软件的产品来弥补微软浏览器安全漏洞的公司），但作为个人来讲，这样是十分冒险的。一个公司，就像我刚才提到的创造移除间谍软件的公司，通常可以应对市场的突然变化，比如说微软突然改进了浏览器安全的缺陷（或者说微软决定要进入移除间谍软件这个市场），但是作为个人来讲，通常没有足够的盈余资金来突然改变自己的职业方向或者职业重心。

> 以特定技术厂商为中心的观点，缺乏远见。
> Vendor-centric views are typically myopic.

供应商的软件实施细节是秘密的，导致以特定技术厂商为中心的观点不能成立。你对某一个软件了解得再多，也会遇

到专业服务障碍。专业服务障碍是由该软件公司人为创造的，在你无法解决某些问题的时候，这个公司就会向你出售他们的支持服务了。有时候这类障碍是故意设立的，有时候是那个公司为了维护产品知识产权（不透露源代码）的副效应。

因此，尽管一心一意地投资在一项特定技术上不是明智的选择，但是如果你必须这么做，那么别选择商业性质的，考虑一下开源的。即使你不想或者不能在工作中利用开源方法，那就把开源作为一个平台，使自己可以对一项技术进行深入学习。例如，你想成为了解 J2EE 应用程序服务器是如何工作的专家，那你要做的不是去致力于研究如何配置和部署一个商业应用程序服务器的细节（毕竟，任何人都会在 config 文件中调整设置，对吗？），你应该去下载一个开源 JBoss 或者 Geronimo 服务器，留出时间来学习这些服务器内部是如何运作的，而不是只学习如何操作。

很快，你就会发现你的观点已经自然而然地转变了。这个 J2EE（或者任何你选择深入学习的的技术）也没什么特别的。现在你已经了解了实施的细节，也知道了工作中有高水平的概念模式。你开始认识到，无论是使用 Java 还是使用其他编程语言或者是平台，分发企业体系结构就是分发企业体系结构。你的视野被拓宽了，你的思想也开放了。比起那些特定厂商的技术，你会发现经过你大脑分类解析的概念和模式更易于扩展，也更能被广泛应用。我要说：“任那些厂商来去自由吧——我知道如何设计一个系统。”

练　习

试着做一个小项目，做两次。第一次尝试使用在家里就能使用的技术；第二次，使用你最惯用的竞争性技术。

10 热爱它，不然就离开它

这听起来像是拉拉队长在大喊加油，目的是刺激你进入一种理想化的疯狂状态。如果你想在工作中做出成绩，就必须对工作充满激情；如果你不在乎这份工作，那后果也会显现出来。

我和妻子刚刚搬到印度班加罗尔的时候，我特别兴奋。在我的职业道路上，第一次期待寻找与我志同道合、对学习充满热情的技术专家。我很期待下班后活跃的生活，我们聚在一起深刻地讨论软件开发方法和技术。我期待看到印度的硅谷被高手们挤得爆满，他们都充满激情地来到这里，寻找软件开发的最高境界。

但是，我看到的大多数人都是来寻找一份工作，很少有人能称得上是热情的开发人员。

这种情景和我的家乡一样。

当然，那个时候我还没有意识到那里和纽约一样。在美国我有一些数据点，但是我一直认为自己在一个不怎样的城市里的一个差劲的公司氛围中工作。我认为这种境况就像我作为一个门外汉第一次踏入 IT 这行时一样。大多数软件开发师都能明白我的意思——我只是没有找到适合自己的工作环境。

通过朋友 Walter 的推荐，我开始在大学的 IT 学院工作。Walter 经常见我用电脑工作，他认为比起大学里那些需要帮助的 IT 工作者，我更能胜任这份工作。可我自己并不这么认为，毕竟自己没受过正式培训。那时，我就是一个爱玩电脑游戏的萨克斯风手。但是，Walter 帮我填了申请表，还安排了面试。面试中基本上没提及一个与技术相关的问题，就这样，我被雇用了，

而且即刻上岗。

刚开始工作的时候，我总是疑神疑鬼担心别人发现我是个江湖骗子。我怕别人说“这个萨克斯风手混在这些接受过培训的专业人士中干嘛？”毕竟，与我一起工作的人都是有高级计算机技术学位的人。而我，拿着个音乐专业的学位，却在这行里滥竽充数。

几周后，真相渐渐浮出水面。那些和我一起工作的人——拥有计算机技术硕士学位的人，他们根本不知道自己在做什么！事实上，有些人居然还在看我如何工作，然后做笔记！

知道真相后，我的第一反应就是假装周围都是傻瓜。毕竟，我没有参加过专业的培训。晚上我在酒吧乐队里演奏，白天沉溺在电脑游戏里。我是因为感兴趣才学习怎么用电脑工作。其实，我学习编程就是因为想自己设计电脑游戏。晚上从喧闹的酒吧回到家里，我还可以用编程软件浏览 Gopher[①]直到天亮。然后上床睡觉，再起床，接着学习，直到不得不出门去演奏。我的生活就是研究我热爱的电脑游戏，吃饭，然后回到 Gopher 或者任何能让我开始工作的编译程序。

> 工作，因为你无法停止工作。
> Work because you couldn't not work.

现在回头看看，那时候我是入了迷了，但这是一件好事。我的创作欲望被点燃了，这和我开始创作古典音乐或者即兴演奏爵士乐的时候差不多。任何我能学到的东西，都让我痴迷。我这么做并不是为了开始一个新的职业，事实上，我在音乐这行的朋友认为我这样是不务正业。但我的痴迷，令我无法自拔。

这就是我和我那些受到过高等教育，但工作表现却欠佳的同事之间的不同——热情。

① Gopher 是一个文件共享系统，目的和万维网相同。随着 Web 的普及，Gopher 的流行大幅降低。

这些人不知道自己为什么在 IT 这行工作。他们偶然进了这行，是因为他们认为做编程收入不错，是因为他们的父母鼓励他们，或者是因为他们上大学时想不到什么更好的专业。不幸的是，他们的工作表现将这一切都揭露得一清二楚。

想想你读过的人物传记或者看过的那些关于伟人的纪录片，虽然这些人都身处不同的领域，但是他们都有一个共同点——痴迷，热情。据报道，伟大的爵士萨克斯风手 John Coltrane 练习非常刻苦，甚至练习到连嘴唇都破皮流血了。

当然，在工作能力上，天赋占了很大的比例。不是每个人都能成为莫扎特或者 Coltrane。但是，我们大可以通过找到自己热爱的工作来摆脱平庸。

一门技术或者一个商业领域可能会使你感到兴奋；相反，或许是某一特定技术或者商业领域拖累了你。也许你更适合一个小团队或者大团队，而你处在不合适自己的组织里；或者是在挑选偏呆板和偏灵活的程序上出了差错。不管是什么原因，想想自己到底适合什么。

短时间内你可能可以伪装，但是缺少热情总会影响你自己和你的工作。

练 习

(1) 找一份自己真正有激情去做的工作。

(2) 下星期一开始，做个简单的日志，坚持两个星期。每个工作日起床的时候，给你的兴奋度打分，分值最高 10 分，最低 1 分。1 分代表你宁愿得病也不想去上班，10 分代表一想到马上就要开始新一天的工作了，你就兴奋，不能再躺在床上 1 分钟了。

两个星期后，检查这个日志。图表中有峰值吗？走向是怎样的？这些点都处在高点还是低点？如果这是一份考卷，那你的平均分是多少？

接下来的两周，每天清晨计划如何在明天得到 10 分。思考你今天要做什么，以便使明天成为你迫不及待要开始工作的一天。每天记录下前一天的兴奋值。如果两周后，这个图表显示的结果还是不尽人意，那或许是时候考虑做一次大的改变了。

做一名机会主义者

——程序员和摄影师

James Duncan Davidson

本文的一开始，我就要告诉读者，我从没考虑过传统的职业规划，我的职业道路是由一个又一个机遇连贯而成的。我的第一个机遇出现在大学时期，我的专业是建筑。十五六岁的时候，我就决定将来要成为一名建筑师，为此我投入了很多。但是上学的时候我痴迷于 BBS，这也为我毕业后的职业选择埋下了种子。我喜欢家里 PC 机上 300 波特的调制解调器，它把我引领到了 Internet，进而让我接触到了 Gopher 和万维网。

我一下就被万维网诱惑住了。我一个接一个地建立个人网站，利用手边的任何技术，有需要的话就自学。那时候，我把这些当作是网络架构的实验。现在听起来这一想法有些好高骛远，甚至有些傻，但这就是万维网刚出现时我们的生活，我们在想象未来将会是什么样子。

当然，Internet 的未来不是在实验室里建造的，它发生在世界经济中。很快，有一家新成立的公司找到了我，他们在为像希尔顿和商业服务管理监督局这样的单位制作网站。他们看了我的公共网站，很显然我的技术正是他们所需要的。这份工作的薪水在当时来看高得有些离谱。我想我就大干一番，存点钱，几年之后还可以再回到学校。这样，我接受了这份工作。

那年是 1995 年。我当时并不知道事情的发展会有多快，也不知道自己愿意学点什么样的新东西。

希尔顿酒店网站的第一版中有实时预定的布局，在帮助建造这个网站的时候，我学到了如何使用各种各样的服务器来建造网站。我从学徒做起，几个月后，已经可以创造我自己的服务器结构了。回头看看，这一切似乎有些荒谬。但在那时，这是必需的。我看到了一个机遇，抓住了它，最大程度地利用它，然后按照需要重新改造自己。

这件事引发了下面的故事。1997 年，我到 JavaSoft 去做服务器软件。几年后，我在这个公司的工作以负责 Servlet 规范为结束。但很遗憾，这是个投资不足的项目，而且没有一个团队能帮助我工作，比如建立一个新的参考实现。但我并没有因此而停止，我开始从头建立起一个新的实现，这就是后来发行的 JavaServer 网络开发工具包。可能很多人现在已经不记得这个软件了，但是大多数在服务器上使用 Java 的人都应该知道接下来发行的 Tomcat。这个软件是通过 Apache 软件基金会和它的一个老搭档 Ant 发行的。发行背后的故事可以写成一本书了。我可以满意地说，这一切都是因为我能够最大程度地利用每一个机遇。

除了从一名学建筑的学生变成了一名计算机技术人员之外，我曾经还是一名摄影师。祖母教了我摄影的基本知识，父母也鼓励我。所以从我记事起，我就随身带着一个照相机。这曾是我生活中很重要的一部分。在离开 Sun 公司后，我为自己编写了一些软件，这些软件都没有发行，它们都是我用来处理照片的。

2005 年，在我从一名建筑系的学生成为一名软件开发师 10 年后，一个朋友给我打电话，他在 O'Reilly 会议集团工作，他们需要一个摄影师，问我有没有兴趣。我答应了。但是我做的远不止简单拍几张照片。我发狂般地工作，参与每个重要会议，向 Flickr

上传照片来提供最及时的更新。我再次被他们聘请，四年中围绕着它我拓展了客户群，创建了自己的业务。

写这篇文章的时候，我还在时不时地编程，也为一些客户做点软件方面的工作。但是，近期我工作的重点是摄影，我几乎成了一名专业的摄影师。或许有一天，这一状况会改变，毕竟，谁也无法预测未来。

但有一点我可以肯定，我是一个机会主义者。当某事引起了我的兴趣，让我感动兴奋，我会立刻奔向它，不惜一切努力使之成功。通常这都需要学习新的技术，开发新的潜能。有人会觉得学习新东西是缓慢费力的过程，但是我喜欢学习新事物。毕竟，新技术可以让你展开新的工作。我永远不会用自己的技术来定位自己，而是用我已经做过的和我将要做的事情来定位自己。技术只是一种做事的方法。

第 2 章

在产品上投资



作为一名萨克斯手，我颇具天赋，我可不是在吹牛，先来听我解释我为什么这么说。我做全职萨克斯手的时候，演出很多。忙的时候，一天就会有两到三场。一天内，我会在早午餐的时候演奏爵士乐，婚礼晚宴的时候演奏舞曲，然后在聚会或者在酒吧里演奏 R&B。我之所以说自己有天赋，是因为我发现自己在工作中不断学习，并且不断进步。我对音调很敏感，光听就能学习新的曲子，还能即兴创作。

但是作为一名萨克斯手，我确实没花什么心思。什么事情都是唾手可得，我觉得自己也很满足。在乐队里我是典型的关键人物，所以我的同伴也没让我感到过有压力。

年轻的时候我进步很快，我没有发现我在慢慢地停滞不前，但是随着演奏 R&B 次数的增多，我演奏的曲子听上去越来越相似。每个晚上我演奏的曲调都如出一辙。我即兴演奏的独奏曲也都是重复前晚或者以前演出时演奏过的曲调。现在想想，我觉得不只是我一个人这样，我周围的专业音乐氛围就是这样。我们从不挑战自己，观众也没有让我们感受到挑战。你见过观众因为萨克斯风手持续吹奏一个音超过 30 秒而鼓掌欢呼吗？

最近这些年来，我一直让自己忙忙碌碌，这样就无暇顾及音乐。很长一段时间我完全没有碰过我的萨克斯风和吉他。直到最近，我才认识到音乐在我生活中的重要性，认真地重新拾起了我的萨克斯风和吉他。这次，我在当地没有音乐界的朋友，没有时间全职演奏，由于生疏，演奏得也不再那么出色了。这次，我只是为自己而演奏。

我不敢确定是不是因为自己更成熟了，也或许是更有智慧了，这次我发现只是花一点小小的心思就起了很大的作用。我并没有拿出乐器就开始演奏，这次我必须自己独自演奏。我听音乐然后记录下想要学习的技巧。比如，我想做到 Phil Woods 在高音萨克斯风独奏时的每一个细节；我还想学习

Prince在Purple Rain专辑中的 “Let’s Go Grazy”这首曲子中是怎么让吉他发出那样尖锐的声音的。

事实证明，在天赋的帮助下，我只要投资几个小时就能使“这些我总想要做的事情”成为现实。而且随着我投入的时间增多，能力也随之提高。学会一个技巧就有助于学会下一个，攻克一个障碍练习部分，就推动我想要攻克下一个。

就这样专注练习了几个月后，我演奏得比以往任何时候都出色，甚至胜过我做职业萨克斯风手的时候。我在自己的能力（或者说是兴趣）上的投资，彻底击败了那个认为一切都是因为天赋的我。

这就是一个强有力的证据。如果你想要拥有一份可以在职场上出售的产品，一份让你具有竞争力的与众不同的产品，你就必须要在这个产品上投资。在商业中，有想法，有天赋的人很多。只有向这件产品中投入心血、汗水、眼泪和资金，才能使它真正具有价值。

在本章中，我们将会就职业投资策略展开讨论，探讨如何选择某种技巧和技术来投资，以及不同的投资方法。本章，工作将真正开始。

11 学习钓鱼

老子曾说："授人以鱼，不如授人以渔。"这句话说得非常好。但是老子没有提到如果这个人不想学习怎么钓鱼，第二天又向你要另一条鱼怎么办。有老师也要有学生才能构成教育，但是大部分人都不愿意成为学生。

> 要主动问，不要等着别人来告诉你！
> Don't wait to be told. Ask!

那么在软件界，"鱼"是什么呢？它是一个过程，一个你使用某种工具、某种技术的某一个方面，或者获取你工作领域的某一特定信息的过程，它是如何从你团队的源控制系统中找到一个特定的子目录，或者是启动一个应用服务器用于开发。大多数人认为这些细节都不是什么问题。你可能会想，会有人来处理这些问题的。源代码管理系统出现问题了，那就去找创建它的人来处理。基础架构小组知道怎么在你和客户之间建立防火墙，所以出现问题时，就给他们发封邮件让他们处理。

谁会希望自己总是任人摆布？如果你想要雇佣某人来为你工作，你会希望这个人总是要受那些专家的支配吗？我不愿意。我想要的是一个能够自立的员工。

很明显，你的出发点应该是学习如何使用你所处行业的工具。以源代码管理为例，它是一个非常强大的工具，它的功能中很重要的一点就是使开发人员的工作更具效率。你不应该把它视为你放置代码的地点。它是你整个开发过程中的重要组成部分。"工作权威资料库"很重要，但别让它束缚你。一个独立的开发人员如果检测过一个项目，那么在存储器中同时存在最后和最好的两个版本的时候，他很容易就能辨别出这两个版本之间的区别。或者你需要对最终发行的代码进行漏洞修复。深夜里你突然间发现一个致命的系统漏洞，你总不会想打电话向别人求助，找到正确的版本然后再开始解决问题吧。这种问题存在于 IDE、操作系统，以及任何一个基础环节中。

你使用的技术平台也是同样的重要。打个比方，你正在使用 J2EE 开发应用系统。你知道应该设置不同的类别，端口和部署描述文件，但你知道为什么要这么做么？你知道这些设置是如何被使用的么？当你启动一个 J2EE 容器时，到底发生了什么？你可能不是一名应用服务器开发人员，但是了解这些可以帮助你为一个平台编写安全代码，当出现问题时，也可以很快地解决问题。

一种最简单的懒办法就是使用向导为你生成代码，这在 Windows 开发中尤其盛行，这些开发工具使很多工作变得非常简单。但是同时，它也使很多 Windows 开发员根本不知道向导背后的代码是如何工作的，向导的工作永远是一个迷。别误会我的意思——正确的代码生成工具是一种非常有用的工具。例如，代码生成器可以将高级 C#代码翻译成能在.NET 运行的字节代码。你当然不愿意自己编写这些字节代码。但是，站在更高的层次上看，使用向导会让你的知识浅薄，你永远也不懂得向导做的工作。

或许你会很容易就忽视掉这行的“鱼”。如果你在一家抵押贷款公司工作，在估算利率的时候，你可以向专家请教，或者你可以自己学习怎么估算。但是与客户的互动是非常重要的，清楚地了解客户的要求比似懂非懂然后自己填写细节要好得多。如果你真正了解你所工作的行业的详情，那会大大提高你的工作效率。你不需要懂得每个行业的每一细则，但你至少应该了解最基础的规则。与我合作过的出色的软件业人士，很多人都成了客户所处行业的专家，甚至比他们的客户更了解那个行业，这样他们的工作成果显然更好。忽视行业性质的人，往往会犯低级错误，只要懂得最基本的行业知识，这些错误是完全可以避免的。而且，与那些了解行业知识的开发人员相比，这些人的工作效率要低得多（最终导致成本的增加）。

对我们软件开发人员来说，老子所讲的道理，或许可以翻译成“要一鱼，食一日；要人授以渔，终身受用。”但是，请注意，不要要求别人来教你，要自己主动学习。

练 习

(1) 如何与为什么？——在你读书或者工作的时候，想一想工作中你不完全懂的问题。你可以问自己这两个问题：它是如何工作的？为什么会发生这种情况？

对这两个问题，你可能给不出答案，但是只要你问了，就会形成一种新的思维模式，也使你更加关注自己的工作环境。IIS 服务器是如何通过向 ASP.NET 发送请求来结束工作的？为什么我必须要为我的 EJB 应用程序生成这些接口和部署描述文件？我的编译程序如何处理动态和静态链接？如果店主住在蒙大纳，为什么计税的方法就不同了？

当然，这些问题的答案很有可能会引发对这些问题的新一轮探索。当在这个“如何和为什么”的环节中你无法再深入了，那就证明你已经达到目的了。

(2)“提示”时间——在你的工具箱里挑选一种非常重要却经常被忽视的工具。可能是你的版本控制系统，可能是一个你广泛使用却只知皮毛的库，也可能是你用来编程的编辑器。

选定了工具后，每天花一点时间学习这项工具的新知识，帮助你提高工作效率，或者能让你更好地掌控开发环境。比如，你可以选择操作 GNU Bourne Again Shell (bash)。当你的思绪游离出手头的工作时，你可以上网查询关于使用 bash 的提示，而不是装载 Slashdot。很快，你就可以找到有用的资源来教你如何使用 shell。现在，有了新的诀窍，你就可以利用一系列“如何和为什么”的问题来深入研究它的核心了。

12 学习行业是如何运转的

在上一章，我们讨论了仔细选择业务领域的重要性。不能小看业务领域的知识，它可以决定雇用方是否会选中你，而且也会让你在工作中赢来阵阵掌声。开始学习某一行业的细则之前，应该确定所做的选择是适合自己并且适合市场现状的。

但是，有一种知识既不属于技术范畴也不是特定于某一行业的，而且也不会很快就过时，它就是财务基本知识。不管你在哪一个行业工作——制造业、医疗部门、公益机构或者教育系统都是一个行业。行业本身就是一门你必须要学习的知识。

我记得当我还是一个年轻的程序员时，参加一个员工会议。我呆滞地盯着一位公司的高管，他向我们展示着一组又一组的数据图表。我永远也不会直接与这位高管共事，他所展示的数据与我完全没有关系。我低声嘀咕着“我就想回到办公桌前，完成我手头上的应用程序功能。”我的团队成员们坐在一起，就像是长途旅行车上局促不安的孩子。我们没人关心会议内容，根本不知道为什么叫我们来参加这个会议。我们责怪这个无能的经理召开这个会议，浪费我们的时间。

> **只有了解了一个行业后，你才能创造性地有所建树。**
>
> *You can't creatively help a business until you know how it works.*

现在回想起来，我才知道我们当时是多么无知。我们来这个公司工作，目的就是为它赚钱或者省钱，但我们根本就不懂这行是怎么赚钱的。更糟糕的是，我们根本就不认为这是我们应该知道的知识。作为程序员和系统管理员，我们认为自己正在做的工作就是我们应该做的。但是，如果连这行是怎么赚钱的都不知道，又怎么能创造性地帮助公司赚取利润呢？

上一段中有一个词——创造性地——是关键所在。没错，我们是 IT 专家，这也是公司付钱雇我们的原因。有了合适的项目和领导班子，我们就应该努力做这个业务，根本不需要了解这行是怎么运作的就能为它提供价值。这些想法看似有道理。

但是，有创造性地增加价值需要全面地了解你所工作的行业环境。在商业世界，我们常常听到“账本底线”这个词。但到底有多少人真正理解“账本底线”是什么，以及什么能对它产生作用？更重要的是，又有多少人知道自己怎么做才能对这个“账本底线”起到有利的作用呢？你的组织是赔本还是盈利（你是为其创造了利润还是给它造成了损失）？

了解你公司的财务运作可以让你做出有意义的转变，而不是茫然无知地专注于某一事情，却主观地认为这样就是对的。

练 习

(1) 通读一本基础商业教程，一本 MBA 教程是不错的选择。我推荐一本非常有用的书 *The Ten-Day MBA*[Sil99]。你真的可以在 10 天内读完，占用不了多少时间。

(2) 找一个人带你到公司的财务部门看看，并请他们向你讲解财务状况（如果你的公司不介意与员工分享这些信息）。

(3) 听完财务状况讲解后，再向他们复述。

(4) 弄明白为什么“账本底线”要被称为“账本底线”。

13 寻找良师

爵士音乐文化中很重要的一部分就是寻找良师。在爵士乐这行，年轻的乐手从师于一名有经验的乐手是很普遍的。这些有经验的乐手收他们为徒，传授他们爵士乐的真谛。不止如此，师傅们除了教授知识外，还会成为职业顾问和生活顾问，年轻的乐手会向他们征求意见。同时，年轻的乐手对师傅非常忠实，会围绕着他们的良师建造一个狂热的乐迷网络。

这样乐手与乐手之间就有了关联，而乐手也通过这样的关系网找到了工作机会。爵士乐文化是一种围绕师徒关系的自组织文化和习俗。这个系统起到了很好的效果，甚至让人怀疑这是由某个组织机构领导的。

> 可以依赖别人，但要确保这个人是靠得住的。
>
> It's OK to depend on someone. Just make sure it's the right person.

在传统的职场世界里（特别是IT业），我们很少能求助于他人。依赖别人被看作是脆弱的象征。我们害怕承认自己不够完美。竞争无处不在，只有强大的人才能在竞争中生存下来。遗憾的是，这种观念导致了“师徒机制”严重不发达。如果你去问爵士乐手：“你的师父是谁？”他们每个人都会有一个答案。但是如果向程序员提出同样的问题，在美国，他们的反应会是：“什么？”

以前情况不是这样的。西方历史中职业教育的繁盛可以追溯到中世纪。比起音乐领域的人，技术工作者受到的专业培训更强大也更正式。年轻人在开始职业生涯的时候，都会做学徒，从师于某个有名望的技术工作者。他们得到的工资很少，但换来的却是向这名大师学习的特权，而这名大师的任务就是把学徒训练成像他自己一样技艺高超的人。

一名良师最首要也是最重要的任务就是做一个榜样。直到亲眼见识到某人突破你所熟悉的极限时，你才知道什么才是一切皆有可能。榜样的作用就

是定义何为“好”。作为一名棋手，战胜家里人可能会让你感觉不错，但是如果你和一名参加大赛的选手较量，你就会知道原来下棋是一项如此深奥的游戏；当你和一名大师下棋时，你又会得到另一个启示。如果你只是一直和你家里人下棋并且战胜他们，你可能会觉得自己是个高手；没有榜样，就没有动力进步。

良师还可以将你的学习过程形成体系。在上一章中，我们说过选择在哪种技术和行业领域中投资时，你会有很多的选择。有时候，太多的选择会让你不知所措。按常理说，前进总比静止不动要好得多。良师可以帮助你削减这些选择，避免你白费精力。

我刚开始做一名系统支持的时候，我紧紧抓住一个叫 Ken 的人，它是我们学校网络管理员之一。一次我们学校的 NetWare 网络系统出了问题，想要进入学校数字图书馆的学生的电脑都会受到危害，Ken 帮我解决了这个大难题。在那之后，他就再没给过我什么启发（他也没有试图给）。当我让他为我指明怎样才能更有见识，更加自立的时候，他给了我一个很简单的答案：潜心钻研目录服务功能，习惯 UNIX 变体，掌握一种脚本语言。

在无数的技术中，他为我挑选了 3 种。这个被我视为权威的人，非常自信地为我指明了道路，我开始按照他的建议学习这 3 种技术。我的职业道路就是建立在这 3 种技术的基础之上的，直到现在我做的事情都与这 3 种技术相关。这倒不是说 Ken 为我指明的方向是绝对正确的——毕竟天下没有绝对正确的答案。重要的是他缩小了要掌握的技术范围，这样我就可以学习技术，取得进步，而不是停止不前。

良师还是值得信任的朋友，他们可以观察并判断你做出的决定和取得的进步。如果你是一名程序员，你可以把你的代码拿给他们看询问他们的意见。如果你计划在公司或者本地用户组会议上做一次演示，你可以先在良师面前做一次演示，从他们那里得到回馈。良师是值得你信赖的人，你甚至可以问

他们："作为职业程序员来说，我与其他人有什么不同？"你知道他们不仅会帮你分析，还会帮助你取得进步。

最后，就像在爵士乐这个行业里一样，你不仅建立了对你良师的依赖和责任，反过来也一样。当我帮助某人时，我就是在这个人的成功上投资。在他的职业道路上，我向我认为对的方向轻轻地推了他一把；如果这个人沿着这条路走取得了成功，那这也是我的成功。

这样就激励了老师帮助他的学生取得成功。通常，经验丰富的成功人士都会受到一些重要人物的尊重。这个良师就成了你与这个人际关系网之间的桥梁。这种桥梁作用是不能被小看的。毕竟，有句老话是"有本事不如认对人。"

这种师徒关系的形式不重要。不需要特别清楚地要求某人成为自己的老师（当然你要这么做也不是坏事）。事实上，你的导师可能并不知道他在扮演着这个角色。重要的是你要有可以信赖钦佩的人，他可以帮助你做出职业导向，帮助你磨练技术。

练 习

指导自己——我们都希望有人主动来教我们，但事实是我们很难在自己周围找到这么个人。所以要学会自己做自己的良师。

想想在你工作的领域中你最钦佩谁。大都数人在不同阶段都会有一个名单。这个人可能是工作中的同事，或者你很欣赏这个人做出的某项成果。列出这个榜样的 10 种特性，这些特性必须都是视他为榜样的理由。这些特性可能是某一特定的技术方面，比如技术广度或者某一特定领域的知识深度。或者是他们的某种人格魅力，比如他们是很好的团队协调者，或者他们的言辞总是具有吸引力。

现在，把这些特征按重要性的升序排列，1 是最不重要的，10 是最重要

的。这样你就提炼出了一个特征列表，这些特征都是你钦佩并认为重要的。这就是赶上你选择的榜样的方法。但是，要先专注于哪一项呢？

想象你的榜样在此项上会给你打多少分（10 分最高），在这个名单上加一列，将分数标注在特征旁边。尽量站在你所选择的榜样的角度上，以第三者的身份来评定自己。

一切准备就绪后，再在整个列表的最后再加一列以记录你的最终得分。如果你给某种特性打 10 分，即最重要的特性，然后对这项你自己的表现给出的分值是 3 分，那这项你的最终分数就是 7 分。得出各项的最终分数后，按降序排列，这样你就能分出改进的先后顺序了。

从这个列表中的前两三项开始，列出你要改进的具体事项。

14 做一名良师

要想真正学点东西，可以试试向别人传授这些知识。清楚知道自己是否对某一知识真正理解的最好的办法就是把你的理解讲给别人听，让他们明白。这个简单的方法是公认的帮助你理清思绪的灵丹妙药。在软件开发这行，大家都知道软件开发师经常会对着宠物或者什么无生命的物体讲述如何解决一个问题。

> 想要弄明白自己是不是真正懂得某一知识，那就把它讲给其他人听。
>
> *To find out whether you really know something, try teaching it to someone else.*

Marin Fowler①曾经在班加罗尔的一次开发师讲座上说，当他想要真正学懂一些知识的时候，就把它们写出来。Marin Fowler 是著名的软件开发师和作家。如果我们把他当成是一名作家，远程传授知识，那他就是这个行业中最负盛名、最有影响力的老师。

我们通过传授知识学习。这听起来有些有悖常理，因为我们希望老师对他要传授的知识了解得一清二楚。当然，我不是在说我们通过向别人讲授的同时就能学到新的真理——要是这样的话，那这些真理是从哪来的？但是，知道真理并不意味着同样知晓它们的前因后果。这种深层次的理解才是可以通过教授学到的。当要阐明复杂问题的时候，我们会用简单一些的事例打比方。我们会弄明白为什么一个类比看起来似乎是可行的，但其实是行不通的，而另一个类比看起来不行，却是行得通的。当你向别人讲授的时候，你就必须回答这些你可能从未想过的问题。通过讲授，我们的那些知识死角就会暴露出来。

所以，就像你找到了一个好的导师，做别人的导师也会使你受益不浅。

---

① 他和我没有任何关系。

做别人的导师也会产生积极的社交效果。一组重叠的导师和学生创造出一个紧密且紧密的社会关系网。在职业关系网中，比起那些熟人来说，导师和学生的关系是最紧密的。当你处于这一关系中时，你们彼此就是忠诚的。在这种关系网中，可以很好地解决难题或者是寻找工作。

> 做导师不会下岗。
> Mentors tend not to get laid off.

你不应该低估帮助别人的感觉——那感觉棒极了！如果你能随时想着别人的利益，那这就是你在用你的技术来帮助别人。当今的经济环境很不稳定，帮助别人这项工作是不会使你下岗的，而且这份工作带给你的收入是不会随着通货膨胀而贬值的。

寻找学生的方法不是你声称自己是权威，而是使自己具备真才实学并且有耐心愿意与别人分享你的知识。如果你并不是某方面的绝对权威，也不要惊慌。有时候你只需要具备某方面的经验，然后去帮助那些比你经验少的人。想想看自己有没有这样的机会去帮助别人。

比如，你可能已经做过大量的 PHP 工作，你就可以参加当地 PHP 用户组会议，向那些比你经验少的使用者提供帮助，帮他们解决具体的问题。或者，你目前还没有这种面对面的辅导机会，很简单，你可以在网络留言簿或者聊天系统中回答问题。但是要记住，这种通过网络建立的师生关系远远比不上面对面的师生关系。

你无需去建立一个正式的师生关系，就从帮助别人开始，好处会自然而然随之而来。

练 习

(1) 找一个你可以帮助的人。你可以在公司找一个比自己资历浅经验少的人，比如一名实习生。你也可以去当地大学的计算机信息技术学院去做志

愿者，辅导一名大学生。

(2) 找一个网络论坛，并挑选一个主题。开始帮助别人。慢慢地，你就会因为愿意助人以及有能力帮助别人学习而在这个论坛里出名。

15 练习，练习，再练习

当我还是一名学习音乐的学生时，我经常彻夜在音乐学院教学楼里练习。训练室的墙壁很薄，我演奏的声音经常被一些十分难听的演奏声吞没。这并不是说我们学校的乐手都很差，正好相反，他们在练习。

当你在练习的时候，演奏出来音乐或许总是难听的。如果你在练习的时候，总能演奏出悦耳的音符，那就证明你一直无法突破自己的极限。这就是练习的意义所在。运动也是一个道理。运动员在训练的时候总是将自己推到极限处，这样他们才能在比赛中突破自己的极限。他们让丑陋的东西都暴露在平时的练习中——而不是真正的比赛中。

在计算机这行，经常会有开发师突破自己取得进步。但是，很多时候都是因为他们本来就不胜任自己正在做的工作。我们这个行业习惯于在工作中练习。你能想象一名专业的乐手，站在舞台上，演奏出来的却是大学训练室里那些难以入耳的声音吗？这肯定是让人难以忍受的。音乐家是通过公开表演而赚取报酬——是表演，而不是练习。同样，如果功夫高手或者拳击手在比赛中表现得疲惫不堪，那他在这项运动中也没什么发展。

我们应该寻找时间练习。在西方，与外包给那些国外团队的工作相比，我们经常把相对高水平的编程工作交给当地的开发人员。我们要在质量上与他们做竞争，就不能把工作当作练习来对待。我们要在提高技艺上面投资。

几年前，效仿练习演奏，我开始试着进行编程练习。第一条规则就是练习开发的东西绝对不能是我想要使用的。我不想图方便，仓促地达到目的。所以我开始编写我用不着的程序。

我没有走捷径，但是在练习中我发现很多想法都不能实现，这让我很泄

气。尽管我尽可能地把它当做工作来好好做，那些设计和编码却不能达到我的期望。

现在回想起来，当时那种窘迫的感觉是个好的迹象。我编写的代码并不是完全没有亮点。我在开发大脑，突破自己的编程极限。就像练习吹奏萨克斯风时，如果练习的时候演奏出来的都是悦耳的音乐，那我知道我根本没达到练习的目的。同样，如果练习时编出来的程序都是很棒的，那我就是在发挥我的正常水平，而没有接近我的极限——好的练习应该让我接近自己的极限。

在极限处练习。
Practice at your limits.

那么，我们怎样才能知道要练习什么呢？怎样才能达到极限呢？作为一名软件开发师，就如何练习这个话题，可以单独出本书来讨论。作为开始，我还是会借鉴我作为爵士乐手的经验。我把爵士乐演奏练习分解成以下几个类别（由于很多人不是乐手，所以我把它们简化了）：

- 身体与协调
- 视奏
- 即兴演奏

这个框架可以成为软件开发师练习的一种方法。

身体与协调：乐手要进行乐器演奏技巧训练：发声、身体协调（比如练习手指的灵巧度）、速度和精准度。这些都是非常重要的练习项目。

那对于我们软件开发师来说，这些基础练习又是什么呢？那些初级编程语言中，有没有你基本不怎么使用的？比如，你选择的编程语言支持正则表达吗？在很多的编程环境中，正则表达式非常强大但却很少被使用。大多数开发员即使可以用到它，也不去使用，因为他们的技术没达到那个水平。结果，创造出了很多不必要的字符串解析编码，并且不得不延续使用。

这条规则也同样适用于 API 和库。乐手们常说，对于技术，你要是没达到手到擒来的地步，那它们真能帮到你的时候你也想不起来它们。这就需要尝试深入研究，比如，在你选择的编程环境中，多线程编程是如何工作的。或者 stream 库、网络编程 API，甚至是一切可用的处理集合和列表的工具。大部分现代编程语言都提供了丰富和强大的库，但是软件开发师们只学习了其中的一小部分子集。如果他们掌握了如何使用一整套工具，那么编程的效率就会提高很多。

视奏：对一名专业录音乐手来说，首要能力就是能够在第一时间完美地视奏。我曾经为 Blockbuster 公司（音像租售公司）的圣诞乐曲专辑中吹奏萨克斯风。在一曲大型乐队演奏的快节奏的乐曲中，我演奏了序曲和第二段高音部分。当磁带开始滚动的时候，我才第一次看见乐谱。我演奏了一遍序曲，又演奏了一遍第二部分。任何失误都会使整个乐团重新再演奏一遍，这样就会浪费时间，并且增加工作室的租金。

作为软件开发师，视奏又是什么呢？是需求规格，还是设计？开源社区是找到用来练习的代码的绝佳场所。有没有哪个开源软件是你最喜欢的？你可以给它加个功能。挑选一个你想要用来练习的软件，浏览它的待办事项，给你自己规定时间来实现这个新的功能（或者至少决定要实现这个功能需要哪些步骤）。

选择好功能之后，下载源代码然后开始开发。怎么知道要看哪里？有什么好方法在一组重要的代码中理出头绪？又要从哪里开始呢？

你可以经常进行这种练习，而且这种练习的周期不会持续很长。你并不一定要去实现这个功能，就把它当做是一个开始，练习真正的目的是以最快速度读懂你正在看的代码。但是一定要确保这个软件与你平时工作时使用的软件不同。要寻找不同风格、不同编程语言的软件进行练习。记录下是如何使整个过程增加或者降低难度。你使用了哪些方法帮助你理解这些代码？面

对复杂的函数层次，你是以什么为线索，让调用栈有迹可循，带领自己穿梭其间呢？

即兴创作：即兴创作就是在某种结构或者限制的基础上创造新的东西。作为程序员，我发现自己往往在压力之下可以即兴创作。“哦，不！无线网络应用服务器出问题了，我们收不到命令了！”这情景常常发生在最具创造力、最即兴的编程过程中，比如通过无线网络从二进制日志文件中手动重发数据包来修复丢失的数据。没有人特意来为你做这些工作，特别是在这么紧急的关头。这种优秀并且迅速的编程能力在正确的关头发挥作用是非常重要的。

训练思维敏捷和提高即兴编码技术的好方法是通过自我限制的方式来练习。选择一个简单的程序，试着来限制你的编程过程。我最喜欢做的就是编写一个程序来显示那首老掉牙的歌曲“99 Bottles of Beer on the Wall”的歌词。如何能编出一个程序而不做任何变量赋值？或者在保证正确显示歌词的前提下，这个程序最小能做到多小？再加一个限制，你最快用多久能编出这个程序？可以使用定时器，练习编写一个难却小的程序。

这只是一种极限视角的练习方法。你可以从任何学科找到练习的对象，从视觉艺术到僧侣信仰。最重要的是找到你所需要的来进行练习，并且确保你不是在工作中练习。你必须要找出时间来练习，这是你的责任。

练 习

(1) Topcoder——Topcoder.com 是一个很早就存在的编程竞技网站。你可以注册然后通过线上竞赛赢得奖励。就算你对竞争没兴趣，Topcoder 还为你提供了一个练习室，里面有很多可以练习的问题。现在就去注册尝试一下吧。

(2) Code Kata——Dave Thomas 是《程序员修炼之道》的作者之一，他接受编程练习这个想法，并使之实用化。他创造了一系列的很小却深具启发性的练习，被称之为 Code Kata，程序员可以使用他们选择的编程语言来做这些练习。每一个 Kata 都针对某一特定技术或者思考过程，这样程序员的思维就可以更加灵活。

这本书的印刷过程中，Dave 已经创造出了 21 个 Kata，你可以在他的博客上免费使用（http://codekata.pragprog.com/）。在这个博客上，你还可以看到通讯名单的链接，以及别人解决这个问题的方法和相关讨论。

你的挑战：练习这 21 个 Kata，并撰写使用 Kata 练习的日记（或者是博客）。当你完成这 21 个 Kata 后，开发你自己的 Kata，并与别人分享。

16 做事的方法

“软件开发”不是一个名词，而是一个动词词组，它是一个创作的过程。当我们编码的时候，开发的步骤与开发的成果一样重要。如果你忽视开发的步骤，那就有可能会误工，创作出的产品有某种缺陷，或者什么也创作不出来。这些后果都会引起用户的不满。

幸运的是，人们在开发一些优质的软件的过程中（以及其他产品）投入了很多心思。这些现有的技术被编录成“方法论”。你可以在网络上或者书店里找到大量这方面的书籍。

遗憾的是，大部分软件开发师并没有从这些优秀的资源中受益。对一个团队中的大多数成员来说，步骤是事后要想的问题。在他们的字典里，“方法论”这个词等同于报告和冗长的会议。很多时候，某种方法是经理强加给他们的。

经理直觉上知道他们需要遵循某种步骤，但却不清楚他们有哪些选择。结果，他们依然延续 20 世纪 80 年代的办事程序，只不过在其中添加了符合年代特征的时髦用语（现在是 Agile 软件公司），然后经理再把这个办事程序传递给他的团队。当团队里的软件开发师自己成为经理之后，还是会重复这个相同的步骤，直到有人打破这个循环，试着证明哪些可行哪些不可行为止。

你肯定会想一定有某种更好的软件开发方法。事实上对于大多数团队来说，的确有更好的方法。

如果你是一名程序员、测试员或者是软件设计师，你可能会认为开发的步骤不是你的工作内容。就你的公司而言，你或许是正确的。但是，它通常

不是任何人的责任。如果非要把这个工作分配给某个人，那可能就得单成立个“步骤小组”或者类似的某个不相连的机构。事实上，一个成功的软件开发步骤，必须是由使用它的人来参与制定的——类似你这种人。

要想让自己找到对这个步骤的归属感，最好的方法就是亲自来操作。如果你的机构中没有工作步骤，那研究方法论对你可能会有帮助。吃午餐的时候，和你的同事讨论开发软件中会碰到的问题，以及选择某种规范步骤或许会缓和这些问题。把你们选择的步骤整合成一个计划，然后得到每个团队成员的认同，并开始执行你的计划。

> 想要拥有自己的步骤，那就执行它。
>
> *If you want to feel you own a process, help implement it.*

你工作的环境中，步骤也有可能是由上级传达下来的。当这个文件到达开发团队，可能已经变了味，或者被重新解释成了新的模样。像中国的一个游戏“耳边传话”[1]，要传递的话从第一个人传到最后一个人那里时，可能会完全走了模样。那这就是一个开始行动的机会。研究这个方法，试着向你的团队和管理部门说明它最初的意思。你无法反对这个上级下达的步骤，那不妨通过正确的阐述使之运作起来。

方法论听起来像某种时髦的名词，很空洞。但是，这对软件开发步骤的研究会有所帮助——即使是研究你并不需要去做的事情。如果你很擅长软件开发过程，那你就拥有了一个更有力的论证来证明你的团队如何才能更好地工作。

即使有很多角度的方法论可供选择，一个公司也不可能完全照搬某个方法。没关系，一个能使你的团队工作更有效率，帮助你们生产出更好的产品的步骤就是最好的选择。

---

① http://en.wikipedia.org/wiki/Chinese_whispers。

方法论：不只是给电脑发烧友的

尽管项目管理不一定与软件开发方法相关，但你可能会发现自己已经开始接触公司的项目管理技巧。在这个行业里，人们使用着大量的项目管理方法。其中最著名的要属项目管理学院所出的“*Project Management Book of Knowledge*[①]”了（本书以其著名的认证制度闻名）。

另一个非针对软件的高质量方法论是六西格玛[②]。由 General Electric 和 Motorola 这样的公司引领，六西格玛方法强调测量、对过程的分析以及生产效率和用户的满意度。六西格玛严谨的系统方法直接适用于程序员的日常工作。

① http://www.pmi.org/。

② http://www.isixsigma.com/。

唯一找到项目管理与软件管理混合方法的办法是研究可用的选择，挑选出适用于你和你团队的方法，从真正的实践中不断提炼总结。

如果你不能沿着这个程序走，那你就很难生产出产品。比起找到一个做软件开发的人来说，想要找到一个能够设计出软件开发步骤的人要难得多。所以，学习软件开发步骤的知识可以祝你一臂之力。

练　习

选择一个软件开发方法论，并且挑选一本有关此方法论的书，登陆相关网站，加入一个讨论此问题的联络组。从批判的角度来研究此方法论。此方法论的优势和弱点在哪里？在你的工作中，执行它的障碍是什么？研究完一个，再换一个继续研究。对比他们的优势和弱点，想一想如何能把它们结合起来。

17 站在巨人的肩膀上

作为爵士乐手，我会花大量的时间听音乐。我听音乐，不只是在阅读或者开车的时候把它当做背景音乐，而是真正在聆听。如果爵士即兴创作就是演奏你听到的琴弦拨动出的乐曲，那么认真听音乐是一种非常重要的灵感源泉，也可以帮助你判断什么是可行的，什么是不可行的；什么是出色的，什么是一般的。

从现有程序中得到领悟。
Mine existing code for insights.

爵士唱片的丰富历史是一套惊人的知识体系，供相关人士聆听。因此，对爵士乐手来说，听音乐不是一种被动的行为，而是一种学习。更重要的是，能够理解你所听的音乐是一种日积月累培养出的能力。我音乐圈里的朋友确实在做这种听音乐的练习。我们会举办这种听音乐的聚会，在聚会上爵士乐手聚在一起听音乐然后讨论。有时候我们其中一个人会放一张即兴独奏唱片，其他人根据这个风格判断出这个即兴演奏的人是谁。

当然，爵士乐手没什么特别的。古典音乐作曲家也这么做，小说家和诗人，雕刻家和画家同样如此。研究大师的作品是成为大师的一个重要步骤。

听爵士唱片的时候，我们会讨论即兴独奏乐手传递音乐主旨的技巧。“哇！你能听出来末尾处他是用什么方式回旋的吗？”或者，“在第三个八拍中，他滞后于节拍的演奏方法真奇怪。”通过这样的讨论，我们发现这些技巧，并吸收精华，下次即兴表演时可以拿来试用。

从这方面讲，软件设计和编程与音乐是相同的。我们可以从大量的现有程序中寻找模式和技巧。设计模式运动（见 *Design Patterns*①[GHJV95]）关

① 本书中文版《设计模式：可复用面向对象软件的基础》已由机械工业出版社出版。
——编者注

注可重复使用解决软件开发问题的方法。设计模式使现存程序的研究正式化，使大量专业软件工程师得以进行这项工作。但设计模式也只是我们读程序编码的时候可供使用的知识中的一个小子集。

那么其他程序员是如何系统地解决某一特定问题呢？其他人是如何有策略地使用变量、函数和结构命名的？如果想在一种新的语言中执行 Jabber 即时信息协议，该如何做？在处理 UNIX 和 Windows 系统的进程间通信时，又能有什么创新的方法呢？通过学习现有的程序，这些问题都可以迎刃而解。

> **用现有程序来反思自己的程序。**
> ***Use existing code to reflect on your own capabilities.***

比找出某一特定问题的解决方法更重要的是，将现有的程序当做一面放大镜来检查你自己的编程风格和能力。就像每当我听 John Coltrane 的唱片时，都会提醒自己作为一名萨克斯手，我站在技术阶梯的哪一层，欣赏编程大师的作品也会起到同样的作用。尽管如此，这不仅仅是说我们自己要谦卑。当你读这些程序的时候，你会发现某些你可能永远也不会去实践的工作，甚至是你想都没想过的。他为什么要这样做？他是怎么想的？他这样做的目的又是什么呢？即使你读到的是不怎么样的程序，通过批判的角度来研究它，你也会有所收获。

这种向已出版的作品学习的行为在艺术界更加容易，因为一幅画或者一首乐曲的背后并没有隐藏的源代码。你听到了一首曲子或者看到了一件艺术品，那你就能从中学习。谢天谢地，我们软件开发师可以通过开源软件来研究无数可使用的现存软件。

开源软件数量很多，所以要想全都研究完也是不可能的。当然这其中肯定存在一些不好的开源项目，但也有不少是非常优秀的。有些开源编码甚至可以在任何编程语言中完成任意软件能够完成的任何任务。

当你以一种批判的视角去看这些程序的时候，你就会开始培养自己的品味，就像你对音乐、艺术和文学的品味一样。不同的风格和技巧可能会使你觉得好笑、惊讶、气愤，或者像我所希望的，让你觉得有挑战。如果你是真的在认真学习它们，在设计范例碰到问题的时候，你会更具创造力。就像在艺术这行，你学习别人的习惯时，就会发展出自己独特的风格。

阅读这些程序的另一个作用就是能让你知道哪些方法是已经存在的。当你碰到一个待解决的问题时，你可能会记起曾经在处理这样或者那样的项目时，看到一个执行 MIME 类型的库。如果这个方法是正确的，那你就会因为使用已经存在的方法而节省了很多时间，同时也为你的公司节省了开支。当你意识到在软件这个行业中，正因为我们一遍又一遍地重复发明轮子（发明这个词用在这里太笼统了）而浪费了多少金钱时，你会大吃一惊。

牛顿说过："我看得更远，是因为我站在巨人的肩膀上。"像牛顿一样的智者都清楚地知道我们能从先人身上学到很多东西。做一个像牛顿一样的人。

练 习

(1) 选择一个项目，像读书一样研读并且做笔记。归纳出好的方面和坏的方面。发表一篇评论。至少找到一个你可以借鉴的技巧或者模式。再找到至少一处缺点，提醒自己在开发软件的时候不要犯这类错误。

(2) 找到一些志同道合的人，每个月聚会一次。每次聚会由一个人提出一段代码——2 行或者 200 行都可以。分解它，然后讨论这段代码背后的东西。思考做出这个程序所做的决定，权衡没有包含在这个程序中的代码。

18 在工作中，将自己自动化

作为 IT 界的管理人士，我期望把项目承包给低成本（国外）的咨询公司，但同时又坚信雇佣最便宜的开发师通常并不能把项目的成本降到最低。这个斗争在我的职业生涯中一直没有停止过。我与 IT 总监或者副主管进行过很多次激烈的争论，我强烈要求雇佣一些真正优秀的开发师而不是雇佣一大批成本低但技术水平也低的写代码的人。

遗憾的是，每次我都是话还没说完就得闭嘴。并不是因为我的观点有问题（显而易见没问题！），而是我很难证明自己是正确的。从成本来看，能够证明他们的观点是正确的唯一证据，就是较低的按小时计算的开销确实有利于公司节约雇佣成本。

想象一个你能想到的任何软件开发项目。如果这个项目要在三个月内完成，那需要雇用多少程序员？5 个还是 6 个？（别嫌我烦）如果同样一个项目要在两个月内完成呢？如何节省出一个月的时间？

IT 部门主管的标准答案是——要想加快项目进程，那就增加程序员的数量。这是不对的，但是大家都这么认为。如果你可以通过增加人手来加快一个项目的进程，这样推断的话，也就是说越多的人就意味着更高的工作效率。

要想达到相同目标，其实有很多种方法。如果目标是提高软件开发的效率，你可以：

- 找到工作效率更高的人来做这个项目
- 找更多的人来做这个项目
- 自动化工作

由于目前为止我们还无法真正衡量软件开发的生产率，因而也就很难证明一个人比另一个人工作的速度快。所以财务经理关注的是一小时支付的薪酬高低。这就引出了一个简单的公式，适用于一个规定的时间段：

$$\text{生产率} = \frac{\text{项目数量}}{\text{程序员数量} \times \text{平均时薪}}$$

在某些情况下，也许可以确切地计算出软件项目的投资收益。但多数情况下，怎么计算项目个数，怎么统计需求数量，都没有既定、统一的标准，而且这些标准依情况不同无法简单地套用。

所以雇佣工作效率高的程序员这个方法是很难证明对错的，我们也不会鼓励增加廉价程序员这种方法。那么剩下的方法就只有自动化了。

还记得20世纪80年代美国失业率极高的时候，我们不仅责备外来打工的人，还责怪机器，特别是计算机。工厂里都安装了大型的臂状机器，他们比人类产量高而且比人类精准，这样人类与他们好像根本就没什么可比性。每个人都非常低落，除了这些机器人的发明者。

把你的公司想象成一家为小公司建立网站的公司。基本上你所做的就是一遍又一遍地创立相同的网站，网站上有联系方式、产品概述、购物车以及相关事物。你可以雇一小部分工作效率高的程序员创建这个网站，然后雇一组廉价的程序员一遍又一遍地手动重复所有工作，或者你可以创建一个系统来自动生成这个网站。

我们向财务经理的计算公式中插入一些（虚构的）数字，就得到了图2-1中的等式。

自动化属于我们这行的 DNA。但是某些原因致使我们不自动化我们的工作。你怎么能够有理有据地比外包团队更加迅速和廉价地开发出更好的软

件呢？制造机器人，把你自己变为自动化。

效率高的程序员

$$\frac{5}{3\times \$80}=0.02$$

廉价程序员

$$\frac{5}{20\times \$12}=0.02$$

一名程序员+机器手臂

$$\frac{5}{1\times \$80}=0.06$$

图 2-1 生产率对比

练 习

(1) 挑选一个你经常重复做的工作，为它编写一个代码生成器。从简单的部分做起。不要管这个代码生成器的重复使用率，只确保这个代码生成器可以节省你的时间。

想办法提高生成代码的抽象等级。

(2) 研究模型驱动架构（MDA）。尝试一些可以使用的工具。看看工作中哪里可以使用 MDA 概念。想想如何用你日常使用的工具来应用 MDA 概念。

从 IT 顾问到常务董事

——Enterprise Corp 常务董事 Vik Chadha

我从通用公司的 IT 顾问，成为 bCatalyst（拥有 5 百万基金的企业加速器）公司的一名创业合伙人，这并不是我预先设想好的职业规划。

那我又是怎样离开一个拥有上万名员工的《财富》排名前五强的企业，到一个刚刚开始创业并且在一项还不成熟的高科技项目投资的公司中工作的呢？现在回头看看，这过程中出现了一些零散的点，我把这些点连接起来，发现了一些重要的图形，在这里我想和大家分享，希望你们可以有所借鉴。

在弗吉尼亚理工大学取得电子信息工程硕士学位后，我加入了 GE 公司，成为了一名 IT 顾问。因特网的商业用途开始盛行，我接手了几个项目，这些项目都是为这个强大无比的平台和它的技术支持所设计的，我与公司的每个团队合作——从 IT 财务团队到技术服务团队，再到销售人力自动化团队，最后到销售数据仓库团队。我喜欢研究最新技术，然后运用它们来解决商业难题。

但是，研究尖端技术也不总是有趣的。我们在研究这些还不成熟的技术过程中，常常碰壁，花费了大量的时间和精力为使用者调试产品。从客户的角度看，不管一种技术看上去多么出色，只有当它能够解决真正紧急的难题并且让他们真正受益时，它才算得上出色。久而久之，这一观点使我从以技术为中心的思维方式向以解决方案为中心的思维方式转变。几年之后在 bCatalyst 公司评估新技术启动的时候，这种新的思维方式充分体现了它的价值。

尽管我在通用工作得非常开心，但总觉得还是少了点什么。作为 IT 专业人士，我觉得自己一直以来都只是在一个方面发展自己的技术，没有机会学习公司到底是怎么运作的，利润是怎么产生的，又是怎样继续积累的。我没有因此而气馁，我决定行动起来，更多地去了解商业知识，以及学习如何做一名企业家。我没有参加过任何商业课程的学习，要想学习这里面的来龙去脉，唯一的方法就是实践，反复试验，从失败中学习。

我以前一个想要创业的室友，也是我的好朋友 Raj Hajela，我妻子 Vidya，还有我，我们三个人集思广益，试图发掘出现在市场的需求。我们想从电子商务中寻找商机，但却不想销售任何产品。我们有一些艺术相关的背景知识，并且对艺术都非常感兴趣，因为每一件艺术品本质上都是独一无二的。我叔叔就是一名艺术工作者，以此养家糊口，只是生活非常艰辛。通过调查研究，我们得出结论——大部分艺术家的生活都是这样清贫。所以，我们决定要创建一个平台来解决这个问题。通过这个平台，帮助艺术家展示推销他们的作品，并与他们的资助人保持联系。这样，我们就发布了 Passion4Art.com，开始寻找艺术家加入我们的网站，把他们的电子作品放到网站上来展示。当我们与 1 000 名艺术家签订合约，并且制作了他们自己的网站后，我们开始相信自己所做的工作是有意义的，并且开始寻找资金。

那时候（大概是 1999 年），一家叫做 eMazing.com 的公司就不同的话题提供每日小贴士。我们想或许我们可以和他们合作（我们提供艺术家，他们提供分销渠道）提供每日艺术小贴士。他们公司的一名高管与我们会面，赞成我们的观点，并决定试行。

我们告诉他为了建造基础结构，我们正在寻找注资，他提出可以把我们的商业企划案寄给城里的一家新企业加速器公司——bCatalyst。

几天后，我们接到 bCatalyst 执行总裁 Keith Williams 的电话，他想要与我们面对面地谈谈，进一步了解我们的项目，我们相当地兴奋。我后来才知道他们是通过一个可靠的消息提供者对我们进行了深入地了解，直到那时我才知道这有多重要。所以，当你试图得到风险投资商的青睐时，尽力找到一个热心的推荐人，因为这是得到面谈机会的最好办法。

通过和 Keith 的几次交谈，我们发现这次合作是可行的。不过当时网络泡沫开始出现，在这上面投资对他们来说不是个好时机。在最后一次会面上，他们表明非常喜欢我们的团队，但是这不足以让他们出资。如果我们可以再想出一个吸引他们的点子，他们就会毫不犹豫地支持我们。我问他们是否真想与我们合作，还是礼貌地拒绝我们。他们向我们保证，言出必行。

后来我又约 Keith 见面，告诉他我愿意从通用辞职，在接下来的几个月全职与他们工作，共同寻找其他的机会。这样一来，他们既不必做出长期的承诺（类似在你买下一个程序前，我们提供试用的机会），也降低了我们的风险。我说服他们相信我愿意离开通用，而且是在没有后路的情况下离开，这就说明了我的决心，说明我愿意全身心地投入到这个项目中。

接下来的一年里，每天我们都会与不同的团队见面，他们向我们推销自己的点子试图得到我们的支持，我注意到我们向每个公司提问时出现了一套新模式。

我把这套提问汇编成了一个列表并与你分享，希望以后你需要风险投资商赞助的时候能用得上。（详见 http://www.enterprise-corp.com/resources/assessment.htm）。

那一年里我在 bCatalyst 公司学到的技能成就了今天我在 Enterprise Corp 公司常务董事的职位。过去 7 年中，我与 100 多个公司合作过，帮助他们募集到 7 500 多万的资金。如果那时候我没有冒险尝试新事物，就无法得到这么丰富的经验。这条路上的蜿蜒曲折是不可或缺的部分。读者朋友们，我希望我的故事可以激励你们找到自己独特的道路——一条可以充分发挥你能力的道路。

第3章

执 行



你已经在正确的市场上做了所有正确的投资。比如你已经成为了一个以服务为导向的无线 Pizza 速递应用程序的架构专家，而且 Pizza 速递这行开始前所未有地繁荣起来。别太自鸣得意，别忘了我们所说的都是准备工作，这些准备都是为了——执行。现在要动真格的了。

除非你特别幸运，不然只凭聪明，或者只凭你是最新技术的专家，是找不到工作的。你为之工作的公司是要盈利的，你的工作就是帮助这个机构实现这个目标。所有的斟酌思考和准备工作都是为了让你在工作中崭露头角，在公司里青云直上。

就像第 1 章第 9 节中那个想成为 J2EE 架构师的家伙一样，很多人在公司中都找不到自己的身份。我的意思是，我首先是一个程序员，然后才是帮一家世界 500 强的公司卖洗碗机的人，对吧？我是一名应用程序架构师——不是一家大公司的雇员。如果把软件看成一件工艺品，这就不太奇怪了。我们挑选的艺术品不总是与使用它的领域一致。为国防承包商设计办公室的建筑师仍然还是一个建筑师——而不是一个国防承包商。

这个有关身份的观察视角制造了一些细小的问题，因为我们宏观上的目标可能与微观责任相冲突。如果建筑师为国防承包商建造的办公室有功能缺陷，那他设计的东西就没有价值。不管他的建筑多么具有艺术美感，他都不是一名好的建筑师。

雇主付给我们报酬是要我们创造价值，我们必须要把书本上的东西付诸实践。在成功的道路上，你的能力在没有得到实践之前不能带你走太远。冲过终点的才是最终的胜者——即那些完成工作的人。

把一件事情做完的感觉是很美妙的。一旦你找到了心里的那团火，你就不想停止了。现在，让我们开始点燃这团热火吧。

19 就是现在

想象你在参加一场比赛，冠军奖金是 10 万美金。第一个开发出能被客户所接受的新程序的团队将获得此奖金。你和你的工作团队报名参加了此次竞赛。此次比赛的时间为一个周末时间。为了赢得比赛，你们设计的程序必须要通过充分测试，符合一套最小集的规定功能。开始时间为周六早上，下周一早上为结束时间。周一早上第一个完成程序的团队获得胜利。如果到周一没有团队能交出完整的程序，那么能执行最多功能的团队获胜。

你很有自信，极其细致地阅读这个程序的功能规则，你发现这个系统不论是在大小，还是在领域上，与你之前做过的程序都大同小异。你的团队成员达成协议，周日中午完成这个任务，但就在那么一瞬间，你突然开始质疑：这个系统，从大小和领域来说，与我们之前做过的很相似，那这个原本要花费几周才能完成的工作怎么可能在一个周末完成呢？

但是当比赛开始的铃声响起，你们开始工作后，你发现团队是可以达到既定目标的。团队按照常规工作，努力地设计出一个又一个功能，修正错误，快速做出设计决定，一步一步地完成工作。这感觉很好。在公司的设计评审和状况汇报会议上，你就经常梦想成立一个小团队，脱离官僚作风，在规定时间内创造出一个新的程序。

很多人都有过这样的梦想。我们很清楚公司里的办事程序拖延了我们的时间。不止如此，我们周围整个环境的办事速度都拖慢了我们的速度。

就现在，我们能做些什么？
What can we do? Right Now?

根据帕金森定律：“工作会自动膨胀到占满所有可用的时间。”可悲的是，即使你不想这样，你还是会掉入陷阱，如果这些工作是你根本就不想做的，这种情况则会更明显。

就比如说这个周末进行的编程比赛，时间非常紧迫，所以你也就没有机会拖延时间。没时间让你熬着迟迟不做出决定，所以你也就不会拖着时间不做决定。你无法躲过这无聊的工作，你知道你必须得尽全力第一时间完成工作，所以你也就没时间觉得这工作无聊了。

帕金森定律是一种经验观察——不是一种无法逃脱的宿命。即使是人为制造的一种紧迫感，也足够使你的效率提高两三倍了。尝试一下，你会体会到的。你可以更快地完成工作。现在就去试验，不要总是只动嘴，不动手。

如果你把自己做的项目当做是一个竞赛，那你就可以更快地完成工作。开始行动，不要总是安于现状，要做推动者。

随时记得问问自己："现在我们能做些什么？"

练 习

查看你的工作日志，看看已经在这上面挂了很长时间的任务，那些已具雏形的项目，或者是某些让你头痛的工作——可能是因为官僚作风，也或者是因为分析瘫痪。

在这其中找出一项你可以在日常工作的空隙时间完成的工作，比如平时你浏览网页的时间，查看邮箱的时间，或者是可以缩短你的午餐时间。把一项以前需要数月完成的工作，在一周内做完。

20 读心术

我以前有个同事叫 Rao，他来自印度南部的一个海滨国家，但他被外派到美国来我们公司工作。Rao 可以按你的要求编写任何程序。你可以让他编写低级的系统程序，也可以让他来编写高级的应用程序。

但 Rao 真正的出色之处在于他做的准备工作。他有一种神秘的能力能预测出你要让他做什么，而这些事情甚至连你自己都没想到呢，他就像变魔术一样。我甚至指责过他跟我耍过什么花样，但是这不太可能。比如我说："Rao，我一直在考虑改变在应用程序框架中封装控制器的方式。如果我们做一点改变，它就能被应用程序使用而不是被网络应用程序使用。你觉得呢？"

他会回答："我已经做完了。你看，已经在版本控制中检测过了。"这种事经常在他身上发生，如果说这是巧合，那只能说 Rao 认真对待我们这个团队所维护的每一个软件，正在做所有可以想到的、与之相关工作。

那个时候，我是公司应用程序架构团队的领导。除了其他的事情，我们主要负责建立并维护公司应用程序的架构。同事们会花费大量时间谈讨软件开发该如何改善公司状况，我们还经常在一起谈论核心基础元素在这些改进中的作用。

通常在这种谈话中，Rao 不怎么出声，但他也没闲着，他在仔细地听。他后来告诉我他能先知的秘密就是他做的事都是我之前已经表达过想要做的事情，只不过我表达这种想法的方式很微妙，甚至连我自己都没意识到我说过这话。

可能就是在等待咖啡煮好的时候，我说过要是能给我们的编码增加点前所未有的灵活性该多棒。如果我经常表达这个想法或者当时说话的语气非常

肯定，那么即使我没有把这项工作增添到团队的待办任务中，Rao也会行动起来，看看这些工作是不是可行，如果执行起来很容易（而且投资不大），那他就会做出来，然后检测。

> 读心术用得好，人们就会信任你。
>
> *The mind-reading trick, if done well, leads to people depending on you.*

你不仅可以把读心术这招用在你的经理身上，也同样适用于应对客户。如果他们给你的暗示准确，那么你就应该能够增加新功能，而这些功能正是他们准备或本该要求你做的，如果他们知道这样是可行的话，增加那些功能。如果客户要求你做什么，你就做什么的话，他们会对你的工作很满意；但如果你做的事情不仅是他们要求你做的，或者你在他们提出要求之前就已经这么做了的话，客户会非常的高兴——但前提是你使用读心术读出的信息是正确的。

使用读心术也是有风险的。就像是一根钢丝，你会尽量避免在这上面行走，除非你在下面设了一张安全的大网。主要的风险及降低风险的方法如下所述。

- 你使用公司的钱来做没人要求你做的工作。要是做错了怎么办？从小事做起。你猜测出来的事情，如果它小到可以在你日常工作的空隙时间完成，你就可以做，这样能使影响最小化。如果你非常想做，那就在业余时间做。
- 每当在系统中增添编码，这个系统复原的可能就会非常小。如果你猜测的工作可能会迫使这个系统偏离原先的特定构建路线，或者在某种情况下限制这个系统的工作，那就不要做。如果变化带来的影响很大，那就需要做出商业决定了，而这个决定往往不是一个程序员能权衡的。
- 也许你的客户确实稍稍提出了要改变某种功能，你按他说的做了，但出乎你的意料，这种改变使系统功能弱化了，而这并不是客户想要的。在猜测出来的工作涉及用户界面的时候，一定要特别小心。

管理员工和项目是一项具有挑战性的工作。那些可以独立带领项目朝着正确的方向发展的员工是非常可贵的，经常加班加点工作的经理们和客户们非常珍惜和感谢这些员工。读心术用得好，会赢得人们的信赖——这也是你职业的发展方向。这项技术是值得开发和加强的。

练 习

Karl Brophey 是这本书较早的评论者，他建议在你开始下一个项目或者维护下一个系统的时候，做些笔记，记录下你认为用户和经理想要让你做的事情。有点创造性，试着从他们的角度来看这个系统。当你记录下这些有可能会被提出的不太明显的功能之后，想一想如何才能最有效地执行它们。想想用户在短时间内想不到的边缘功能。

然后当你接到执行工作的要求时，看看你的列表，你的命中率怎么样？你猜到的这些功能有多少是现在要做的？在做这些工作的时候，你之前思考过的方法用得上么？

21 每日成绩

凭借我们的知识以及我们是优秀的软件开发人士，我们都想相信自己会自然而然地做到日事日毕，能够成为一流的员工。对很少的一部分幸运者来说，这种策略的确会奏效。

但是计划和跟踪工作成绩，对我们每个人都有益。大家都知道如果我们的工作成果超出了经理的预期，那我们就会成为最优秀的员工。但令人惊讶的是，如果把超出经理预期作为一个有价值的目标，很少人有办法能察觉出如何以及什么时候我们超出了老板的预期。

对于大部分值得去做的工作而言，做那些细致和有目的性的工作的员工更容易变得出众。你最后一次做分外的事是什么时候？你的经理知道么？你又怎么能让他知道你做的工作呢？

> 每天都有可汇报的成绩。
> *Have an accomplishment to report every day.*

我同事 James McMurry，同时也是我的好朋友，[①] 他很早的时候就告诉过我，在与系统相关的工作中，他都会精心做好准备以达到最好的工作效果。那时候他的工作经验不多，能有如此深的见解着实让我吃了一惊（或许是他父母给了他提示），我至今仍在使用这一方法。在没有告知他的经理的情况下，他开始记录每天做出的成绩。他的目标是，每天都要有一些出色的工作报告给经理，这包括他关于如何改善这个部门的一些想法和已经做过的工作。

制定目标（每日、每周或者无论什么你能做到的）并且记录下可以彻底改变你工作中表现的工作内容（如图 3-1 所示）。当你开始追寻骄人的业绩

---

① http://www.semanticnoise.com。

时，你自然而然地就会根据你工作领域的商业价值，对你的行为做出评价，将工作划分优先顺序。

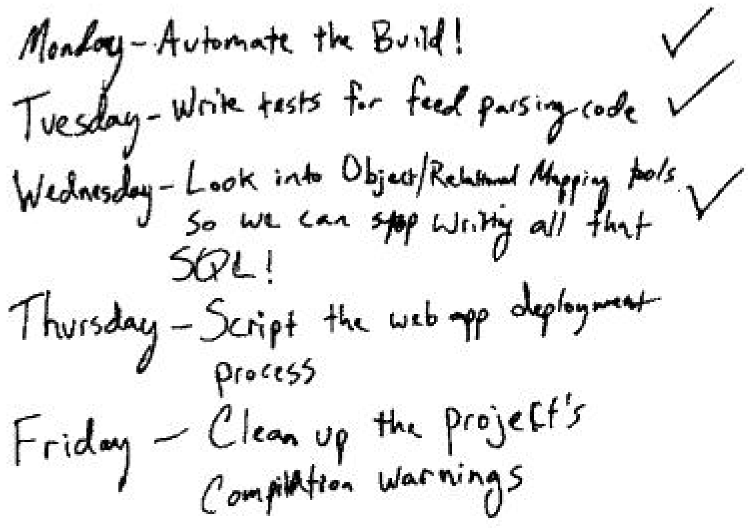


图　3-1

经常记录工作成绩可以确保你不会停滞不前：如果你规定自己每天都要做出一点成绩，那你肯定不能花两个星期的时间精心策划某个“完美”任务。这种思维方法和工作会成为一种习惯。就像一个对单元测试包里绿色横棒（表示测试通过）上瘾的开发人员一样，如果你没能完成今天的成绩，你就会浑身不自在。不必担心每日的进步记录，因为在这一层面工作，更像是一种条件反射，而不是一组需要在项目管理器中做出准备的任务。

练　习

留出半小时时间，拿一支铅笔和一张纸坐在一个安静无人打扰的地方。想想每天你的团队都在忍受的那些吹毛求疵的小问题。把它们写下来。哪些烦人的任务每天都要耽误整个团队好几分钟的时间去讨论，却没人有时间或精力来完成这些任务？

你现在做的项目中，哪些工作是可以自动完成，而你却在手动完成？记录下来。你来构造或者部署一个过程如何？你可以解决哪些问题？又将如何增加成功率呢？记录下这些想法。

给自己实实在在的20分钟，记录下所有的想法，不管是好的还是坏的。一定要做满 20 分钟。列出清单后，在另一张白纸上理出你最感兴趣的（也是最烦人的）5项任务。下星期一，瞄准这个清单上的第一项，任务做点成绩出来。周二，第二项；周三，第三项；以此类推。

22 别忘了你在为谁工作

人们常说“确保你的目标和工作与你公司的目标一致”，这话说起来容易，做起来难，特别是当你是一名程序员，上面有层层部门机构，以至于你根本不知道你公司的业务是什么样的。早一些时候，我在一家大型包裹运送公司工作，我的软件开发架构团队负责这个公司的收入系统。这个公司有一层又一层的等级。在日常工作中，我从没有机会了解包裹运送这个行业。我还记得我们团队每次出席季度会议，都觉得这会议完全与我们无关。“我们庆祝的成功到底是什么？这些数据到底都是什么意思？”

那个时候比起包裹运送这个行业来说，我对建造完善的系统和开源软件编程要感兴趣的多（好吧，我承认——现在我对这些更感兴趣）。但是，如果我当时真的想让自己的工作同公司的重大目标保持一致，说不定就知道该从何下手了。

所以，我们需要与公司的目标一致，也就是说，努力确保自己能够对公司的业绩有影响力，这么说完全没错。但是，说真的，以我们所掌握的信息，很多人都不知道我们应该如何做。只见树木，不见森林。

或许这并不是我们的错误。或许我们对自身的要求太高了。或许想要直接对公司业绩造成影响就像是想把大海里的水煮沸，是不可能完成的任务。所以，我们需要一个更加独立的视角，把整片大海的海水分解成一个一个可以煮沸的小水坑。

可以从你的团队开始。这个团队可能很小而且任务单一。你知道团队的问题是什么，也知道这个团队正在努力改进的问题是提高工作效率、增加收益、降低错误等等。如果你不确定团队的问题和努力的方向，能给你答案的最佳人选就是你的经理。

最后，在结构良好的环境中，经理的目标，就是整个团队的目标。解决了经理的问题，就是解决了整个团队的问题。而且，如果你的经理也是这么想的，那么，你为他解决的问题也就是他上司的问题，就这样一层接一层，直到公司中最高的领导层——执行总监、股东甚至是你的客户。

你做的事情可能很小，但是却是在为完成整个公司的目标出力。这就是你工作的目标，也是你工作的意义所在。

有些人可能不愿意这么做，他会说："我可不想帮他做他的工作。"或者"我上司肯定会把我的功劳据为己有。"

我承认，有这种可能。正如 Lister 和 DeMarco 在 *Peopleware*[DL99]一书中所说，好经理的职责不是"做替补"，即了解整个团队的工作应该怎么做，在出现难题的时候就自己上。好经理的职责应该是为团队设定优先级，确保团队具备完成工作的一切需要，保证团队保持干劲和工作效率，并促使团队最终顺利完成工作。整个团队的工作干得出色，就证明这个经理的工作非常优秀。

> **经理的成功，就是你们的成功。**
> *Your managers' successes are your successes.*

如果说经理的工作是了解和设定工作的优先顺序，而不是亲自去做所有的工作，那么你的工作内容就是去做这些具体工作。你并不是在帮经理做他的工作，而是做自己份内的事情。

你可能担心经理会把你的功劳据为己有，但别忘了你的职业发展前景把握在你经理的手中（至少，在你现在工作的公司中）。在大部分公司中，直接总管负责绩效考核、工资、奖金和升职。所以，你的功劳最终还是在你的经理那里折现了。

记住你是在为谁工作。你不仅要按工作的要求来调整自己，还应该使工作与你自己的需求一致。如果你还可以做到日事日毕，那么你的工作就会朝着好的方向发展了。

练　习

计划一次与经理的会面，目的是了解经理对整个团队下个月、下季度及来年的目标。问问他你能做些什么。会面之后，检查自己的日常工作是否与这个团队目标一致。以这些目标来决定你工作的动向。以这些目标为基础，把你的工作排出先后顺序。

23 安分守己

作为经理，我可以告诉你最棘手的事情就是有个总是想要往上爬的雇员。每次和他一起吃午饭，都会听他说又有谁升职了。这种人总会在办公室里散播点小道消息，死死盯着企业政治，就像对肥皂剧对白一样沉迷。他愤愤不平地完成每天的任务，无休止地抱怨领导部门的无能，总觉得要是自己来做领导工作，肯定能做得更好。还会认为领导们没有发现的他的潜质，是因为他们都太无能了。

这种人认为许多任务根本就不配他来出手做。遇到这些工作的时候，他能躲就躲；实在躲不了的时候也是极不情愿（而且很慢）地去做。他对工作精挑细选，有时候会下意识地挑选那些自认为配得上自己的水平，并且能帮助他实现晋升目标的工作。

> 要有雄心，但不必路人皆知。
> *Be ambitious, but don't wear it on your sleeve.*

可悲的是，这种人总是活在下一个工作中，所以在当前工作中的表现就会很平庸。这种人是我深恶痛疾的，就像让我去修剪草坪，我最烦的事就是割草，这会让我浑身痒痒，而且还会流很多汗。最糟的是，修剪完草坪后，我连本来愿意做的事情都不能做了。你可以雇人来帮你修理草坪，我以前就帮人修剪草坪，但那时我还是名学生。现在我毕业了，当我必须要修理草坪的时候，我怎么办呢？我会马马虎虎地赶紧把它做完。在整个过程中，我会一直想着怎么才能赶紧做完，然后去做我真正愿意做的事情。简言之，修剪草坪时，我极其不认真。

值得庆幸的是，在我修草坪的时候，没人监督我工作，并且没人给我打分（尽管我妻子总是抱怨我再也不管我们家的草坪了）。修完草坪后，如果它不漂亮，那就是我的问题。由于我的工作表现不怎么样，没人拉我去当个“修草

坪的"。在 IT 工作中，同样的行为会为你的职业发展造成毁灭性的影响。上文中我们提到的那个人，你认为管理层会怎么评价他？他们会觉得自己真的小视了这个人的才能，然后提升他吗？他们会为了让他开心而给他涨工资吗？

当然不可能。他工作平平，态度还不端正。就算他真的有很大的潜质，又能怎么样？当下，他并没有展现他可能拥有的资质。公司不是靠可能性来挣钱的。如果不能符合股东们的潜在设想，他们是不会继续投资。而且，就因为他这种不端正的工作态度，他的经理也不会再想培养他。

这就是一个经理的观点。当然，我也不是完全没有罪恶感。某种程度上，我曾经多少也有点像这个人。不过，即使是站在他的角度来看，这样做真的没什么好处。你把所有的时间都花费在想要得到某种东西上，可是欲望同满足是对立的。早上一醒来，你就在想又得去做那份该死的工作，而且没人欣赏你的潜质。你愤愤不平地艰苦工作，一遍又一遍地重复上演着能让你晋升的策略。最近你上司做砸了一件事，你就开始幻想你要是他会如何做得更好。工作对你而言就是拖延生命，除非做到你认为适合自己的职位，并且能按照你的方法办事。

告诉你个秘密：这种感觉永远不会有休止的一天。即使你得到了梦寐以求的升职，很快就又会开始烦躁，觉得你想要的不是这样的工作——你想要的是比这个更高的职位，循环往复。我现在也没坐在最高的职位上，但我有种很强的预感，就算真有那么个职位等着我去做，但只要我一往前看，我肯定会发现自己一直追的其实是一个幻影。这样做，对职业生涯简直就是浪费。

但是，难道我们不应该有雄心壮志吗？如果那些投资者没有雄心壮志和目标，那还会有微软和通用这样的公司吗？

我们当然应该有理想。我不是在鼓吹一种毫无激情的人生观。有目标是好的，期待成功也是好的。但是，回想一下这节开篇我提到的那个被动、一肚子愤懑的人。你认为这个人会成功吗？他的前景看起来并不怎么样。比起只专注在目标上的做法，专注于现在的工作会使你离最终目标更近。

这样做一开始会很难。让你抛弃每天走向成功的动力，这听起来是一个不可能达到的目标，但一旦你尝试这么做了，就会发现它是非常实用的。专注于现在的工作，你就会享受日常工作中的每一个小成功：你工作干得很出色，当出现难题的时候，你就会被人当做专家一样请来解决这个问题。在一个具有凝聚力的团队中，你会成为不可或缺的一员。这种感觉都会让你感到喜悦。但如果你总是头脑不清醒，做白日梦，那就会错失这些成功的喜悦。你会一直在等待那个巨大的成功，却无视每天工作中的那些小成功，但其实正是这些小成功赋予你工作的价值。

专注于现在的工作，不仅会让你更加快乐，也会让你身边的人更加快乐。你的同事、上司和客户都能感觉到。你的工作成绩会反映出你的态度。客观地来讲，放弃你想要成功的期望可以提高你的能力，让你走向成功。

你和你的客户是密切相连的，和你的领导以及那些决策者也是不可分割的，他们会在短期，甚至是长期对你的职业产生重要的影响。菲律宾和印度的程序员没这个优势，但是你有。所以，请安分守己。

练　习

一周内不去想你的职业目标。写下你当下工作的目标。别去想下一步要做什么，想想你想从手头的工作中得到什么，你又能做出怎样出色的成绩？制定一个既有战略性又具实用性的计划。以完成当前工作为目标，用一周的时间来执行这份计划。

午餐或者休息时，在与同事的交谈中，避免涉及任何关于职业发展或者公司政策这类内容，专注于手头工作的目标。

这周的最后一天，把你的进度和你制定的工作目标作对比。就现在的工作而言，你认为需要完成的任务中，有多少还没有完成？你又是如何评价自己已经完成的工作？制定下周的工作计划，坚持下去。

24 今天我能把工作做到多好？

做好工作得到赞赏是有益的。尽管大多数人天生都知道这个道理，但是我们总是纵容自己过分挑剔，连我们要在何处何时不嫌麻烦地施展过人本领都要精挑细选。我们热衷于给市场部制作下一个重大项目的设计，当出现明显的重大问题时，我们还会第一时间冲过去解决问题，因为我们非常清楚这是出名的大好时机，甚至不惜痛苦不堪地熬夜工作。越糟糕的环境越能激发出我们最优秀的一面。

在遭遇最艰难的系统问题和逾期仍未完成的任务时，这种兴奋的陶醉情绪使我保持清醒，工作非常有效率。为什么在没有压力的时候，我们就无法如此狂热地工作呢？在处理最无聊烦人的任务时，如果你也可以如此狂热地想要把它做好，那么你的工作表现会有多出色？

> **你能为工作增添多少乐趣？**
> *How much more fun could you make your job?*

最后一个问题这样讲也许会更好，在处理最无聊烦人的工作时，如果你也可以如此狂热地想要把它做好，那么能为你的工作增添多少乐趣？当我们觉得有乐趣的时候，工作就能做得更好。相反，当我们对某个任务毫无兴趣的时候，我们就会觉得无聊，工作结果也就令人失望。

你能把一份无聊的工作变得有趣吗？如果你倒过来看这个问题，答案就会是显而易见的。你为什么会觉得这份工作无聊？为什么不能从一开始就觉得它是有趣的呢？你喜欢的工作和憎恶的工作之间有什么不同？

对大多数软件工程师来说，工作无聊主要出于两个原因。我们热爱的工作是可以让我们充分发挥创造才能的工作。软件开发是具有创造性的，大多数人也正是出于这一点才从事这份工作。而那些我们厌烦的工作往往是我们

认为没什么创造力的工作。想一想，下星期你工作日程中有哪些工作。你不想做的工作很可能就是没什么机会让你发挥想象力的工作，你会觉得这种工作找其他人来做就足够了。

第二个原因是令人厌烦的工作很无聊，并且没有挑战性。我们都愿意去探索、解决别人解决不了的难题。这就像人们不惜冒着生命危险去攀登高峰、蹦极，原因就是我们热爱那些可以证明我们能力的事情。那些无聊的工作通常都不是什么脑力活，难度和倒垃圾差不多。

那么，在每天的工作中我们又如何挑战自己，发挥创造力来应对这些平凡的工作呢（我们 80%的时间都是在做这些工作）？

试试把这些无聊的工作做到 100 分，感觉怎么样？比如，你讨厌做单元测试热爱编程，但不得不去写自动化测试代码，这让你感到十分厌烦。那你可以努力让你的测试做到完美。这样的目标，会让你在工作中有怎样的改变？对于单元测试这个环节来说，怎样才称得上是完美？这和测试覆盖率有关。完美的测试覆盖率是指所有实码的功能都能测到。完美的单元测试同时也是无暇的、可维护的，并且，对于那些无法在另一台计算机上复制的外界因素，它也不会过多地依赖。在对一台新机器进行新的版本控制检测后，这些测试应该能直接运行。当然，所有这些测试都应该 100%的通过。

这就有点难度了。100%的测试覆盖率基本上是不可能完成的任务。为确保这些测试不受外界因素影响而采取的去耦措施，看上去是很有挑战性的。事实上，为了达到这一目标，你一定会更改编码。但如果你能达成这个目标，那这个测试将会是无与伦比的。

我不知道你们怎么看，但至少我觉得这听起来挺有趣的。当然这只是一个凭空想象的范例，但这种思维方法在工作中是可行的。明天就试试看，问问你自己，“我今天能把工作做到多好？”你发现你会更喜欢自己的工作，你的工作也会喜欢上你。[①]

---

① 感谢Andy Hunt的这一想法（http://blog.toolshed.com/2003/07/how_good_a_job_.html）。

练 习

与你的同事展开竞赛，竞赛的内容就是这些无聊的工作任务，看看谁能把这些工作做到最好。如果不喜欢单元测试，那就看看你每天都要接触到的代码，将其中测试断言的总数打印出来，把它钉在办公桌隔断板上。再为整个团队布置一个记分板，获胜的人就能得到某种权利甚至是奖金。 在项目结束后，获胜的人可以在接下来的一周内，把乏味的工作分配给团队中其他成员去做。

25 你的价值是多少

你有没有认真地思考过，公司雇用你到底投资了多少？你知道公司付给你多少薪水。可还有其他的呢？福利津贴、间接管理费用、培训等，这些不会显示在你的工资中。

人们总是会不断地索取。公司给你涨了工资，一段时间内你会感觉很好，但是你很快就会想什么时候再涨工资呢？“如果我的薪水比现在高 10%，那我就能买一个新……”我们都有过这种想法。在某种意义上，实数已经变成了一种抽象概念。问题不是每年多挣 5 000 美金，而是基数要不断地增长。如果我们的工资没有得到让我们满意的增长，我们就会对自己的工作和公司不满。“他们为什么不欣赏我的表现呢？”

像我之前说过的，公司雇用你到底投资了多少？这笔钱肯定大于你的基本工资。为了便于继续讨论，我们大致估算这笔钱相当于你薪水的 2 倍。也就是说，如果你一年的薪水是 6 万美金，那么事实上公司在你身上的开销就是 12 万美金。

这很容易计算，但难计算的部分是：你一年能产出多少效益？你对公司盈利产生的积极影响是什么？既然公司每年在你的身上投资了 12 万美金（按照我们想像的情况），那反过来，你又给公司创造了多少？你为公司节省了多少开支？你为公司又创造了多少营业额？

这些数字是不是也是你工资的两倍？

这个问题是很难找出答案的，因为我们很难把工作中的每个部分都与公司盈利关联起来。这个问题对你来说甚至有些不合理，你可能会说：“我怎么会知道？我只不过是个编程的！”确实是这么回事。你为一个公司工作，

如果你不能为这个公司创造某种真正的价值，那在你身上的投资就是浪费。我们都认为涨工资是理所当然的，这其实是个陷阱。同样地，一个公司每年也可以理所当然地给自己的产品涨价。但是如果消费者认为价格没有什么吸引力，那也就理所当然地不会去购买。

现在对比一下公司在你身上的开销和你的产出，你觉得你应该创造多少利润才能成为一项有价值的投资？虽然我们现在讨论的是一个粗略的数字——你工资的2倍，但你应该能看出点什么来了吧。如果你创造的价值是你工资的两倍，那你的公司也还是会破产。这是投资的好方法吗？

设想一个典型的消费者储蓄账户，以这个账户的利息作参照。利息并不高，但总比没有好。如果让你选择，你会把一年的存款存入一个利息为0%的账户，还是放入一个利息为3%的账户？如果你创造的利润相当于你薪水的2倍，那么你对一个公司的吸引力，就和一个利息为0的账户对你的吸引力是一样的。公司每年在你身上投入了12万美金，而你创造出来的利润却还不足以弥补年通货膨胀率。

我还记得当我刚刚开始有这种想法的时候，自己都有些神经质了。每过一个月，我就会问自己“我这个月创造了多少价值？”后来发展成每周问自己一遍，之后是每天都会问自己“我今天实现自己的价值了么？”

> 问自己“今天实现自己的价值了么？”
>
> Ask, “Was I worth it today?”

你可以把这个问题变得更具体一些，就是问自己你又新增加了多少价值？可以问问你的上司，如何最好地把它量化。有想把它量化的想法就是好的。你怎么做才能帮助公司节省开支？如何提高整个团队的工作效率？考虑一下使用你们软件的终端客户。一旦你开始考虑这些问题，你会非常惊讶地发现你可以找到很多的机会。现在，就开始行动吧。把这个数字记在脑海里：2倍的薪水。在达到这个目标前，千万不要放弃。

练 习

当公司做投资的时候，领导会想尽一切办法确保他们把钱投在了正确的地方。简单对投资回报做出估算（投入 100 美金，得到 120 美金）不足以做出明智的决定。在其他影响因素中，公司还需要考虑通货膨胀、机会成本和风险因素。对于我们这些没进过商业学校的人来说，一般不会想到货币的时间价值。这样说可能有些过于简单：今天的一美金，明年可能就变成不止一美金了，因为今天的一美金可以用来创造更多的钱。

大多数公司都会设立回报率线，在这之下，就不会进行投资。进行投资必须是在确定的时间内并有确定的回报率，否则就不会进行投资。这个数字被称作“最低预期资本回报率”。

找出你公司的最低预期资本回报率，用它来衡量你的工资。你是不是一项好的投资？

26 一桶水中的鹅卵石

如果你从座位上站起来，走出办公室，然后就永远不会回来了，会怎么样？我知道很多程序员都会想象这个情景来安慰自己。起身，走向老板的办公室，提交辞呈。我要让他们看看我有多重要！这种白日梦可以帮你在一天郁闷的工作中得到一丝安慰，但是显然，这并没有什么建设性的意义。

这个情景也不现实。公司里每天都有人提出辞职，大部分人并没有受到挽留，只得离开了。有的人甚至就直接离开公司，连辞呈都没有递交。事实上，一个员工的离开很难对公司造成很大的影响，即便是这个员工处于非常重要的职位，造成的影响也会非常小。对公司来说，你就像一桶水里的一块鹅卵石。当然，因为有这块鹅卵石在，水平面线会上升一点儿。你完成工作，做好本分。但是，如果你把这块鹅卵石从水中拿起来，然后再观察这桶水有什么变化——你基本上是看不到变化的。

我说这些不是要给你泄气。我们都需要觉得自己做出了某些有意义的贡献。但是我们太沉迷于做自己，却忘了身边有很多人和自己一样。公司里的每个人都不自觉地沉迷于自我，大家也只是从自己这扇窗户来审视自己的工作。这样想：如果你明天离开公司，对公司造成的影响（平均来讲）与其他同事离开没什么区别，或者还不及他们离开的影响大。

我曾经为一位信息主管工作，这家公司是全球最具实力的公司之一，他也是公司里最具影响的信息主管。他带领他的团队（我也是其中一员）横扫公司各个奖项，制定了公司所有的 IT 标准。这个家伙一定是发现了某种神奇的万能药水，甚至在解决千年虫的问题上，他都带着这种药水提供了许多免费午餐。

我从这名信息主管那却得到了这样一条建议——永远不要高枕无忧，他

一遍又一遍地重复这句话。他说自己每天早上起来都会提醒自己要清楚地知道说不定哪天自己就被公司解雇了。他会说，可能就是今天。

他的手下对此表示怀疑。不可能有这么一天。一切都很顺利，你已经做得非常出色。

> **小心！别让成功冲昏了头脑。**
> **Beware of being blinded by your own success.**

这是他的观点。谦逊不仅仅是我们需要发扬的一种美德。谦逊能让你更加清楚地审视自己的行为。这名信息主管告诉我们的是，你越是成功，就越有可能犯下重大错误。当你得到很多肯定的时候，你就会很少质疑自己的决定。你使用的方法屡试不爽时，你可能就会忽视可能会有更好的方法。你开始变得傲慢。人一旦傲慢，就会产生盲点。越认为自己无可取代，就越有可能被别人取代（你在公司存在的意义也就越小）。

感觉自己无可替代是一个不好的征兆，特别是对软件开发人员来说。如果你是不可取代的，这或许只是意味着你做工作的方式和别人不一样。尽管我们都希望自己有某种特殊的天赋，但是基本上没有一名软件开发师是无与伦比、盖世无双、无可替代的。

很多程序员都半开玩笑地说要编出无法维护的编码来加强“工作的安全感”，我也确实见过真有程序员尝试这么做，但他们却无可避免地成为了众矢之的。当然，公司的确会害怕最后放走这些人。可是，从根本上说，让人害怕是最糟糕的情况。这种为了让自己无可替代而采取的自我保护的花招造成了你与公司（以及你的同事）之间的敌对关系。在这种关系中，你已经没有立足之地了。

这样说来，想要让自己不可替代，就要建立一种友好的工作关系。我们每个人都不是不可替代的。清楚知道这一点，并努力工作，恰恰会让自己与众不同，并且无意间为自己创造了更多的机会。而且如果你是可以被取代的，那么就没有什么可以阻挡你迈向下一个更好的工作了。

练 习

把你编写并维护的代码，以及你做过的所有工作编成目录。把你的团队完全依赖你完成的工作记录下来。可能你是唯一懂得你们应用程序部署过程的人，或者你编写的一部分代码，其他团队成员无法搞懂它们。

把这些全部列入你的待办事项中。自动化或者分解你的代码，或者为这些代码提供说明，使其他团队成员很容易明白你编写的代码。按这样，做完所有待办事项列表中的工作。主动与你的团队成员和团队领导分享你所做的说明。确保团队成员可以很容易看到这些说明。定期重复这一练习。

27 爱上维护

几年前，我参与了一个建立250人的软件开发中心的项目。从一间空房子开始，到找到合适的雇员填补这个开发中心的每一个机构。在整个建造过程中，我们碰到了一个从未想过的挑战。每个人都想要建立新系统，没人想维护旧系统。我们想要制造出一种充满活力的工作环境，所以如果我们想朝着正确的方向前进，就必须要注意新的雇员想要什么。

人们都喜欢创造。创造出一个产品，就可以把它贴上自己的标签，感觉它是属于我的，我们可以在创造中表达自己的思想。我们认为完成项目是最可见的工作。建造新系统的人是能够得到荣誉的那个人，对吧？这种态度在我之前工作的团队里十分常见，但是这次，同时有这么多软件开发员想要创新，这种情况我还是头一次遇到。

尽管软件开发员都是具有创造力、热爱自由的人，但是整个编程“社会”却是有着等级制度的。程序员想要成为设计师，设计师又想成为架构师，等等。维护工作太普通了，这个职位不像架构师那样可以拿出来在父母和大学同学面前炫耀。

创造力以及升职的机会都是激发因子。但有趣的是，项目工作不见得是产生这两个激发因子的最佳点。

维护工作一般都是涉及老旧的系统和一意孤行的终端用户。因为旧软件常常被认为是已经完成了的工作，所以IT部门通常会致力于降低维护系统的费用，寻找最便宜的方法来维持系统的运行。

这就会导致维护系统的资源非常少，公司也不会投入大量的资金来修复系统。

相反，做项目工作的时候，起始点是一块非常整洁的绿地。在一个运转良好的公司，每个项目都是为了赚钱或者省钱，所以为了完成项目，资金都会是非常充裕的（依情况而定）。在一个新项目中，不会存在旧代码的雷区，程序员不用为了确保旧系统的功能正常而踮着脚尖小心翼翼地编写代码。总而言之，做项目的环境是更为理想的。

如果我给你1 000美金帮我买一杯咖啡，如果你没买来咖啡，手里的钱却少了，我肯定会不开心。如果你给我买回来非常好喝的咖啡，但是却花费了两个小时，我还是会不高兴。另一种情况，如果我让你帮我去买咖啡，没给你一分钱，你却带着咖啡回来了，我会非常感谢你；如果你没能买回咖啡来，我也会非常理解。做项目就好比前者，维护就像后者。

当没有不好的遗留代码作为阻碍，而且也不缺乏资金，我们的上司和客户当然会对我们有更高的要求。做项目的时候，公司会期待项目创造出效益。如果我们没能产出效益，那我们就失败了。公司的发展依靠这些效益，所以他们经常会严密地控制制造出来了什么东西，如何制造以及什么时间制造出来。突然之间，我们的这片创作沃土变成了一场军事行动——我们的每一步行动都受上述因素影响。

> 维护也可以成为自由和创造的沃土。
>
> *Maintenance can be a place of freedom and creativity.*

在维护模式下，我们所期待的就是用最小的开支来维持软件正常运行。没人期待维护团队能做出什么闪光点。通常如果一切顺利的话，客户基本不会过问维护人员的日常工作。修复错误、实现小的功能需求、维持系统运行——这就是你全部的工作。

如果为了修复一个错误，需要在应用程序中重新设计一个子系统呢？这不就是修复错误么，对吧？原来的设计可能很陈旧，整个系统可能到处都是

破窗[1]的迹象。这就是你把自己的重建印章刻在这个测试中的好机会。这个系统到底能有多么出色？在你做了重建工作之后，下次再修复或者改善系统的时候，能节省多少时间？

只要你一直维持系统运行，并且对用户的要求及时做出回应，维护模式就是自由并且具有创造力的。你同时身兼项目领导者、架构师、设计师、程序员和测试员的工作。你可以随心所欲地发挥创造力，而这个系统的成功或者失败都需要由你来承担。

维护系统的时候，你也可以设计更可见的改进。那个已经运行 3 年的网站管理系统可能无法支持现代网络浏览器的最新用户界面功能。如果你既能维持系统运行，又能修复错误，那么就可以帮助用户更好地使用这个系统，你所做的改进就是可见的。为你的用户安装一些他们意想不到的附属装置，和你买花给你妻子惊喜，或者一个孩子在父母外出购物的时候打扫房子的效果是一样的。而且，少了做项目时候的层层级别限制，你会讶异于自己居然可以把一切都做得很好。你的客户也会非常惊喜。

做维护工作还有一个不易被察觉的好处，维护工程师经常有机会和他的客户直接进行交流。这在现在很多项目团队的合约性工作环境下是很少见的。这样，作为维护工程师，会有更多的人认识你，你就有机会在业务上建立更广泛的支持者。你可以利用这个优势来更深入地了解业务的运作情况。比如你全权负责一个商业应用程序，你会经常与终端用户接触以解决他们的问题。这样，你不怎么费劲就可以慢慢像程序的使用者一样非常了解这个程序到底是用来做什么的。商业规则被转变为编码写进了应用程序逻辑中，商业人士往往读不到这些应用程序逻辑。很多时候只有维护工程师才是唯一懂得公司中某个特定商业过程是如何运作的。其他人都没有机会直接接触到这个权威的商业逻辑编码。

① 更多关于“破窗”的信息，请参阅《程序员修炼之道》。

最讽刺的是，项目工作其实就是维护。只要项目团队写下来第一行编码，每个新增加的功能都是被嫁接在现有的代码基础之上。当然，比起维护工作，编写代码涉及的改动相对少，但是基本内容都是一样的，都是在已有代码的基础上增加新功能、修复错误。如果是一名已经完全掌握维护工作、并且致力于把维护工作做得更好的工程师，谁又能比他把这项工作完成得又快又好呢？

练　习

评估、改进、评估——在你所维护的所有重要应用程序和代码中，把所有可以评估程序质量的元素列举出来。比如，应用程序的响应时间、数据处理过程中抛出的未被处理的异常的数量、或者是程序的正常运行或(可运行)时间。不要直接评估程序的质量。支持请求周转时间 (你应对问题的反应时间和解决速度)对你的使用者来说，是非常重要的。

从中挑选最重要的特征，开始对它进行评估。在有了一个基本的测量准线后，确定一个可实现的目标，改进这个应用程序的（或者你自己的）表现以达到这个既定目标。当完成改进工作后，再做一次评估来证实你确实完成了你想要做的改进。如果你确实完成了，那就和你的团队成员及你的客户分享。

再挑选另一个测量标准，做一遍上述工作。完成第一个标准之后，你就会发现这像做游戏一样，很有乐趣，会让你上瘾的。

28 8小时激情燃烧

围绕着极限编程有很多争论，其中一个争论就是它宣称程序员每个星期的工作时间不得超过 40 小时。这可惹恼了那些经理，他们想要尽可能多地压榨员工。这也可能会让程序员感到沮丧。能够连续工作的小时数量已经成为了程序员们大男子气概的象征。

Bob Martin[1]，极限编程组织的杰出人物之一，在遵循 Kent Beck 的理论的基础上，改变了这句话的表述方式，使其更能够被程序员和雇主双方接受。Martin 把一周 40 小时工作时间表述为"8 小时激情燃烧"。也就是说，工作的时候不容一丝松懈，高度集中，以至于连续工作的时间无法超过 8 个小时。

在继续深入讨论之前，你可能会问，为什么要强调减少工作时间呢？这章不是在讨论如何完成工作么？我们讨论的应该是延长工作时间啊？

在工作上，更少的工作时间可以有更高的效率。首要原因是根据极限编程理论，当我们疲惫的时候，思维就会变得没有效率。当我们彻底耗尽能量时，创造力和工作质量就会明显降低。在这种情况下，我们就会开始犯下愚蠢的错误，结果是既浪费时间又浪费金钱。

> 做项目像是马拉松，而不是全速短跑。
>
> *Projects are marathons, not sprints.*

大多数的项目都是一项长期工作。人们不可能按冲刺的速度跑完整个马拉松。增加工作时间，尽管在短期内大大增加工作成果，但是从长远来看，如果你为了多完成些工作每周加班加点工作 80 个小时，你可能会因此犯下大错误，但是修复这个错误的时间可能比加班的时间还要长得多。

---

① http://www.objectmentor.com。

你可以用看待金钱的方式来看待时间。当我还是个青年的时候，挣到的钱还没我现在浪费的多呢，但我还是非常开心。现在的我比小时候有钱得多，花钱也就不再那么精打细算。但那时候，我也可以生存，有地方住、有车开，有东西吃。

我现在也有这些东西，也没有过着奢侈的生活。很明显，当没有钱的时候，我的钱花得更加高效，最终结果都是一样的。

我认为有限的资源更加珍贵，我们会更加高效地利用有限的资源。花钱的时候是这样，安排时间也是一样。想想你上一次一周工作 70 个小时的情况。在第 4 天的时候，你表现如何？你肯定一直都非常努力。但是到第 4 天的时候，在安排时间时就会有些松懈。上午十点半，你会想反正别人下班回家之后我还得再加班几个小时，那不如现在就看一会儿最新的技术新闻吧。

当工作时间非常充裕的时候，工作的时间价值就会降低。如果有 70 个小时，那对你来说，每个小时的价值远比不上你只有 40 小时的情况。

通货膨胀的时候，你需要花费更多的钱买同样的东西。当时间价值下降后，就需要更多的时间工作。Bob Martin 的 8 小时激情燃烧理论给你制造了约束，但同时也为你提供了应对这一约束的策略。你必须要工作和思考，我就剩 8 小时了！快！快！快！有严格的开始时间和结束时间，你就会自然而然地更加高效地安排时间。你会从一组需要今天就要完成的工作开始，将他们按照优先顺序排列，一次性攻克。

8 小时激情燃绕所制造的一种氛围类似于超高效周末，上大学的时候你可能经历过。在参加一门一直没怎么关注的课程考试之前，或者迟迟未动笔的学期论文马上就要到提交的期限了，这一个周末你肯定会特别有效率。不同之处就是这是仓促应考。仓促应考的时候学习效率非常高，时间非常有限所以也就变得非常珍贵。8 小时燃烧激情是一种让你提早进入仓促应考的状

态，而不用喝着含有高咖啡因的可乐不眠不休。

作为脑力工作者，即使我们不坐在电脑前或是办公室里，也可以工作。在和你的伴侣去吃饭的路上，或者你看电影的时候，都可以工作。你和工作如影随形。

在没有努力工作的时候，工作就会成为一种折磨。我可能没完成某个具体工作，或者让工作堆积起来了。这时候工作就会跟着我回家，在我想放松的时候纠缠着我。如果你每天都投入地工作，就会发现工作不再跟着你回家了。不仅是你故意在下班之后不再工作，而且你的大脑思维也允许你休息。

好好规划你的工作时间。减少工作时间，你的收获会更多。当你离开工作一段时间后，才会更喜欢工作。

练 习

确保今晚睡个好觉。明天，吃个早饭，然后正点去上班（比平常早一些更好）。投入地工作 4 个小时，吃午饭，然后再投入地工作 4 小时，确保 4 个小时后你肯定没有力气再做其他的工作了。然后，回家放松。

29 学习如何失败

程序员都知道在开发过程中越早发现软件的问题，软件就会更加完善。单元测试帮助我们尽早发现奇怪的 bugs。越早在自己的代码中发现奇怪的错误，我们就会越高兴。尽管别人会指责这是我们程序员的失误——是我们制造了程序错误——但错误发现得越早越多，就预示着这个软件会更加完善。

我们希望程序故障出现的越早越好，当出现故障的时候，你就可以正确地修复，安置防御措施。在编程过程中，你一定不会忽略程序中出现的小故障。这就是代码与你对话的方式。这些小的故障是加强程序过程中的一部分。因此，当程序出现问题或单元测试失败时我们会添加断言（assertion），停止程序运行。

我们遇到的小故障也让我们知道我们会遇到什么样的失败状况。如果你从来没闯过布雷区，就永远不会知道不能踏上什么地方或者哪个方向不能走。如果你做的软件没有定期向你抱怨，你就不知道危险的故障隐藏在哪里。

此外，带着防御性措施进行编程是很重要的。出现问题的时候，才是考验软件质量的时候。没有建立应急案的程序，不可避免地会发生一些问题。如果程序员没有很好地对没能预见的故障做好防御工作，那么产品的编码中就有可能出现段错误或者蓝屏。

> 每个错误的音调离正确的音调不过一步之遥。
>
> *Every wrong note is but one step away from a right one.*

这一原则也适用于工作。出现问题的时候，才是真正检测工匠手艺的时候。学习如何处理是非常重要的，但也是很难学会的。作为爵士乐即兴演奏者，一个错误的音调离正确的音调不过一步之遥。如果当这个错误的音调突然蹦出来时，演奏者就束手无策了，那这段演奏就失败了。站在舞台上，一边是乐队，另

一边是观众。一个错音足以让一个业余演奏者惊出一身冷汗。就算是大师也会演奏出错误的音调，但是他们补救的技艺很高超，观众听不出来哪里出现了问题。

工作中每个人都会犯错误。这是人类的天性。我们编程中的错误导致用户去查看栈跟踪。严重的设计错误使我们自己陷入困境。或者，更糟糕的是，我们向团队成员、经理和客户表述的都是错误信息。我们错误地宣称自己有能力做什么，或者当团队成员遇到技术问题时，我们无意间就提供了错误的建议，浪费了他们的时间。

因为我们每个人都犯错误，所以知道其他人也会犯错误，就不会对彼此犯下的错误品头论足。不过我们评论的是如何解决那些无法避免的错误。

以下规则适用于解决技术、沟通或者项目管理中出现的错误。

- 发现问题后第一时间提出，不要企图隐瞒错误。因为在软件开发和测试中，越早发现错误，造成的问题就越小。越早发现并且暴露出自己犯下的错误，造成的负面影响就会越小。
- 接受批评。就算你可以找到一只替罪羊，也别动这个念头。即使这不完全是你的问题，你也要承当责任然后继续工作。出现问题后，我们需要的是解决问题的方法，我们的目标是在最短的时间内解决这个问题。在谁来负责这个问题上纠缠不清的后果就是拖延解决问题的时间。
- 提供解决方法。如果你没有想到解决的办法，那就提供一份有计划性的进程。陈述要按照具体可预测的时间顺序。如果你使团队陷入了困境，告诉大家你什么时间能给出一份解决问题的方案。这一情况下，一个具体可实现的目标，即使它非常小、对问题的解决也没有实质意义，也是非常重要的。因为它不仅使状况从坏向好的方向发展，也帮助你重建自己的可信度。

- 寻求帮助。就算是你对问题负全部责任，也不要让自尊心作祟，拒绝别人的帮助，这样只会使情况更糟糕。这个时候，如果你放下自尊，保持一种良好的心态接受团队成员的帮助，那么你工作的伙伴、经理和客户都会欣赏你的表现。很多时候，我们都会对造成的问题产生一种责任感，致使我们承担起过大的重担，结果是历尽千辛却毫无成果，直到有人强制介入。

回想一下你最近一次在餐厅碰到的麻烦。比如你点的菜迟迟没有被端上来，好不容易等来了，却不是你点的食物。面对你的抱怨，服务员是怎样处理的呢？

> 充满压力的时候是赢得忠诚的最好时机。
>
> Stressful times offer the best opportunities to build loyalty.

如果服务员当时找借口或者把错误归到厨师身上，那他的处理方法就是不妥当的；如果他听了你的抱怨后，任凭你饥肠辘辘地坐在那里，却走开重新再把你点的菜单提交上去，这也是不对的。当然，最错误的表现是服务员明明知道这盘菜不是你点的，却还是端到你的面前，并且期望你根本没注意到或者没有任何抱怨。

公司对错误处理方式的不同，会产生不同的结果。问题处理得好，会使客户对其产品更加忠诚，甚至比碰到问题之前，更加信任这家公司。要是处理得不好，那就摧毁了客户的信任。工作中犯错误的时候，要时刻谨记客户的这一心理。

30 说“不”

清楚知道自己无法做到却还依然做的承诺最不容易实现。没错，这听起来再显而易见不过了，但是我们每天都在这么做。我们被逼上了这个位置，又不想让领导失望，所以我们对不可能完成的任务点头了。

> 为了避免失望而说“是”，就是在说谎。
> Saying “yes” to avoid disappointment is just lying.

说“是”是个上瘾却具有毁灭性的习惯。这是个坏习惯，却伪装成好习惯。“能够胜任”的态度和歪曲某人的能力有很大的不同。后者不仅给自己造成麻烦，也会给你所承诺的对象制造麻烦。假设我是你的经理，问你有没有可能在这个月底前修改公司订单履行系统中查询配送的方式，我问此问题的目的很可能就是强调在这个月底前有没有可能。可能是别人问我这个工作在月底前能不能完成。也可能是我们要依靠这个系统做一次大的商业转变。所以，因为你承诺说可以在月底前完成，我就对客户做出了承诺。

这样说“是”虽然并不是恶意的，但和说谎没什么区别。当我们做出承诺的时候，也等于对自己说了谎。毕竟，说“不”的感觉不怎么样。我们生来就总是想要成功。承认自己不能做某事感觉就像是失败了。

人们没有意识到说“是”并不总是正确的答案，“不”也不一定就是错误的答案。而我们都知道的确是这样的。毕竟，没人想得到错误的承诺。

无法说出“不”是印度文化的一部分。在国外外包经验尚浅的公司里经常会发生这种情况。随着时间的增长，你会学着发现不确定的问题，然后问正确的问题。你一次又一次地听到“再给我一天的时间我就能完成”这样的话，自然而然地你就会深入探索。这不仅是在 IT 行业才会出现。我住在印度班加罗尔的时候，在家里等人来给我安装线缆调制解调器，结果安装的人

根本就没出现，这样的情况不止五次。后来我才知道，前三次他们失约是因为这个公司根本就没有负责安装的部门，但是他们却和我约好了时间来给我安装，这样做只是因为不想让我失望。我告诉他们希望这个线缆调制解调器下周可以安装好，所以即使他们非常清楚下周根本就不可能安装上，却还是对我承诺说没问题。

尽管他们的用意是好的，但是产生的结果却是负面的。后来，我要求他们假日来给我安装，并且态度有些蛮横。我根本不会再相信他们说“明天，假日结束后就来安装”这样的承诺。他们一再食言，完全摧毁了我对他们的信任，甚至使我对他们产生了反感。

另一方面，当有人让你去做一个重要的任务，而你说你胜任不了，又会怎样呢？我既管理外包团队，又管理本土团队，我可以告诉你对我来说我更相信说“不”的团队。如果我的团队成员真的无法完成某项工作时，有勇气说“不”，那当他们说“是”的时候我会更信任他们。敢于说“不”的人做出的承诺更可信，也更有分量。如果他们真的完成了自己承诺的目标，当他们说无法达到的时候，我不会质疑。

经常说“是”的人，要么就是天才，要么就是在说谎。后者通常占多数。

在适当地时候说“我不知道”也是很好的。你可能会被问到能否在某一天完成工作，做出承诺之前，需不需要对工作做一些研究。或者，有人问你某种技术是怎么工作的，或者你项目代码中的某部分是如何执行的。就像上面讲到的做出承诺的例子，尽管不知道这些问题的答案就像是一次小的失败，但是，当你宣称知道某事的时候，你的经理和团队成员会更相信你。某个领域内真正的专家，对于他们不知道的事情，总是勇于承认。“我不知道”并不代表我不可靠。

做决定的时候，也需要同样的勇气。经理宣布某一技术决定的时候，他

的团队成员却静静地坐在椅子上，盯着自己的鞋，等待逃出会议室的那一刻，这样他们就可以相互发牢骚了，这样的情景是不是很常见？经理经常会成为《皇帝的新衣》里的主角。每个人都知道这是个错误的决定，却没有一个人敢说出来。我也是个经理，常常要做决定或提出有分量的建议。但是，我不希望自己的雇员都是机器人。那个敢于说出问题所在并提出更好建议的人，才是我最信任的伙伴。

当然也不要过分地说“不”。“能够完成”的态度还是值得欣赏的，有延伸目标也是好的。如果你不能确定是否能做某事，但是你想要尝试一下，那就说出来。“这是一次挑战，但我想要试一试。”这样的答案非常好。当然，有时候，答案就是“是”。

要勇于诚实。

练 习

评论者 Karl Brophey 建议大家记录下做出的每一个承诺。

- 到期日，你要完成什么？
- 你承诺的内容是什么？
- 如果你无法完成，记录下你的想法和你要接受的内容。
- 做出承诺的时候，记录下来。

每天检查这个记录。在你知道不能实现诺言的时候，第一时间与对方沟通。每月检查这个记录——你做出的承诺中，有多少是实现了的？你又有多少次能做出正确的承诺？

31 不要恐慌

我是因为喜欢电脑游戏，才进了电脑编程这一行。从开始下载游戏到我的 Commodore 64 游戏机的时候，我就对这种逼真互动的体验上瘾了。我以前不愿意承认这个事实，但现在我觉得这也没什么丢人的。对我来说，电脑游戏把电脑屏幕上的环境（我猜是操作系统）转变成了一种使我感到舒服并且感兴趣的环境。

我最爱的一款游戏是id软件公司的Doom，尤其是喜欢这款游戏一对一、玩家对抗、死亡模式的部分。玩家通过调制解调器链接或者通过串联链接起来，在一个快节奏的小环境中对抗。我很擅长 Doom 的死亡模式比赛。我常开玩笑说，这个可能是我迄今为止最擅长的事情了。死亡模式无论是从技术上还是心理上都非常复杂——就像是下快棋和击剑的疯狂结合。

和其他技术一样，提高水平的方法就是观察大师是如何操作的。在我玩 Doom 的时候，有个大师级玩家给自己起了个线上名字叫“菜鸟”。“菜鸟”绝对是 Doom 玩家中的冠军。远在北美洲的玩家不惜付昂贵的长途电话费，也要和他对决碰碰运气。这些比赛都会被记录在 Doom 的内置游戏记录系统中。我亲眼见证了每一次比赛。

没过多久，我就发现了他的秘密。他很擅长玩游戏，但是他的成功有一个明显的要诀：他从不会让自己陷入恐慌之中。在 Doom 这一游戏中，很可能一个回合刚开始，就立刻结束了，速度非常快。还记得我第一次玩死亡模式的时候——出生，死亡，出生，死亡。当我终于挣扎着坚持了几秒钟时，发现我一直在毫无目的地乱跑，根本不知道自己跑到哪里了。

“菜鸟”从来不会像我这样。不管境况多艰难，从游戏记录你就能看出来他总是很放松地思考下一步该怎么做。他总是十分注意把当前的情况与整

个比赛关联起来。

想想其他的比赛，特别是在运动竞技中，最出色的选手都具有这个优良素质。事实上，出现在书中、电视剧和电影中我们钦佩的人物也都具有这一品质。英雄从不恐慌。当有原子弹袭击他们的城市，或者飞机坠毁的时候，他们总是可以组织团队，帮助幸存者，与敌人斗智斗勇，或者至少不会流着眼泪被击倒。

英雄从不恐慌。
Heroes never panic.

现实生活中也是一样。不管做出怎样完美的计划，我的职业生涯还是有着一连串的突发事件和灾难。项目完成时间总是一拖再拖，软件应用系统崩溃，浪费我老板的钱，降低他们对我的信任度。我对一个副总说了一些错误的话，这些话本不应该对他讲，结果我得到了一个政治敌人。大部分时候问题都是一个接一个地出现，祸不单行。

在最艰难的时候，我会恐慌。我把自己锁在屋子里，尽最大能力思考。我对这些问题分开独立思考，却没有想到应该要考虑全局。

但是现在回头看这些灾难，没有一个对我和我的职业发展产生长期并且明显的影响。所以，尽管在这些看上去是灾难的情况下我非常地惶恐、沮丧或者苦恼，但它们都不是真正的灾难。

惶恐给了我什么呢？消极地应对这些情况的益处是什么呢？什么都没有。惶恐带给我的就是在最需要我充分发挥自己能力的时候，我却无能为力。

我不得不承认在困难的时候不要惊慌这句话说起来比真正去做要简单得多。就和你告诉别人“要快乐”是一个道理。当然，这是个好的建议，但是怎么才能做到呢？当一切都四分五裂的时候，怎么才能保持镇定，不惊慌？想想我们为什么会慌张，可能会对回答这个问题提供些帮助。

我们惊慌是因为丧失了判断力。出现问题的时候，要想不完全关注问题本身是很难的。某种程度上来说，关注问题本身是解决问题的好方法。但是，这通常会制造问题，不管你碰到的问题多小，看上去都会比它本身更严重。随着问题不断膨胀，压力就会越来越大，我们的大脑就不再运转了。

你认识的最差的电脑使用者是谁？在我看来，可能是我父母或者我妻子的父母中的一人（我知道是谁，但我还没傻到在这说出名字）。想象一下，这个人坐在电脑前面，想要完成一件工作，但是不论他做什么，都不断地跳出报错信息。我们都见过这场景。没经验的电脑使用者很快就会产生挫败情绪，不知所措。他们会忙乱地点击或者拖拽屏幕上的图标，却完全忽视报错信息提供的帮助。最终他们会很惊慌，又弄出一两个新问题来，然后找别人来帮忙。

别觉得我刻薄，我希望你能把我说的这个人换成你所认识的人，他们这样做有点荒谬，是吧？非常可笑。

但是，尽管这非常可笑，这一情景经常出现在日常生活中。当某人碰到问题无所适从的时候，就会惶恐。当项目超过预期完成时间或者我偶然间摧毁了一个系统，又或者我在工作中令客户不满了，我的反应也是这样。只不过情况不同而已。

接下来说说我是如何学习不要惊慌的。当碰到问题的时候，我就产生了那种无助、充满压力的感觉而导致的惊慌，这个时候，我就拿自己与那个挫败的电脑文盲相比，然后嘲笑自己。我以第三者的身份来分析当时的境况，就像我帮助家庭成员解决他们使用 Word 的过程中出现的问题。表面上看起来很难解决的问题突然间就变得简单了；看上去不利的境况突然间也好像不是那么糟糕了。我常常发现解决问题的办法其实很简单，而且非常显而易见，就像错误信息对话框经常会告诉你下一步该如何做。如果你当时阅读了错误信息提示，问题可能就解决了。

练 习

做一份关于恐慌的日记。在惊慌之前克制住它的办法就是当自己的感觉和情绪爆发的时候，要有一种超强的实时意识。我很幸运，通过事后分析自己应对问题时的反应，学会了这点。我不够聪明，没办法在事情发生的时候，分析自己的想法，但是我发现在正常情况下练习分析，使我取得了一些进步，当困难出现的时候，也能更好地做出即时的分析。

更好地分析自己的反应，说和做是两回事。写日记可以给整个过程创建一个结构。每天定时（用日历或者提醒）记录下使你惊慌的境况，即使只让你感觉到一点点小惊慌也要记录下来。每周回顾一次上一周的日记，观察每一个引起惊慌的情况造成的持续影响。这个状况值得惊慌吗？对这种境况最有帮助的反应应该是什么？电影里的英雄人物要是碰到你这种情况，他们会怎么做呢？

这样反复练习之后，你会慢慢发现当你惊慌的时候，会同时开始做这些分析了。当你能够在碰到问题的时候理智地分析引起惊慌的原因，惊慌就会黯然隐退，最终消失。

32 说出来、行动、展示

从不做出承诺最容易导致一事无成。没有截止期限，你也就没有压力或者说是动力来完成某项工作，特别是当任务不是十分有趣的时候。

再差劲的经理也知道制定计划的重要性。对一些开发师来说，听到“计划”这个词就紧张。整天没完没了地与老板开会，制定大量的项目计划，这计划没人理解也没人用过，这也是制造紧张的原因之一。

计划不至于是特别难喝的药，必须要屏住呼吸强迫自己喝下去。计划也可以让你有释放感。当一天有很多事情要做的时候，有了计划就可以让你从混乱中理清思绪，攻克所有工作。

计划也不一定非得是大计划，也不需要有很长的延续性。在文档中做个列表或者一封邮件就足够了。计划也不需要有很长的时间跨度。在一天开始的时候问自己“今天要做什么？”就是很好的开端。我知道很多人都回答不出这个问题，所以他们每天都过得非常疯狂。今天下午找个时间，把明天要完成的事情做个列表，并将它们按先后顺序排列出来，这就是一个很好的开端。人们总会认为自己可以做很多的事情，但请尽量实际一些。

一天的计划，可以非常详细，也可以是泛泛的。我大学时的室友 Chris，他每天早上起来一定要制定一天的计划，就算上课要迟到了，也必须要先做计划。他特别注重对练习钢琴的时间做出计划（他的专业是爵士钢琴）。其实在他选择课程的时候，他的计划已经非常缜密了，但他对课间休息的 15 分钟也做出了计划，他选择那些可以很快完成的练习来填补这段休息时间。他大部分课程的上课地点都在一栋教学楼里，所以本来他课间休息有足够的时间社交或者喝杯饮料，但他没有这样做。我们都坐着等着下一节课开始，Chris 却充分利用这点时间练习音节或者听力。他甚至把时间按三分钟至五

分钟分段，这样就可以在十分钟的时间里安排几个练习了。现在 Chris 是我们这个城市最受尊敬的音乐家之一。当然，天赋在这其中起到了一定的作用，但是我一直认为是他一贯按计划行事成就了他今天的成绩。

现在，你制定了自己的计划，可能不像 Chris 制定得那么详细，但是也足以回答“今天要做什么”这个问题了。明天上班的时候，拿出这份计划，从第一项工作开始做起，午餐回来再继续按照计划行事，努力在下班之前完成计划上所列的事项。

当你完成这份计划上所有的工作之后，在上面写上“完成”两个大字，这样你会很开心。一天结束之前，再看看写着“完成”两个大字的计划，感受这份成功的喜悦。通过这份计划，你不仅知道了今天要做什么事情，也清楚地看到自己已经做了什么。

如果你没有完成这份计划上的所有事情，不要担心，继续把今天未完成的工作添加到明天的计划中（如果这些工作仍然需要完成），因为一天的工作量是有限的。这是个刺激过程。通过制定计划，你的生活就会由一连串小的胜利组成，每获得一个小胜利都会激励你去实现下一个成功。你会发现，这样做不仅让你能够清楚地看到自己到底做成了哪些工作，而且比起你毫无计划地工作，按照计划行事其实可以完成更多的工作。

当每天的计划按节奏完成之后，你可能就会想制定一周甚至是一个月的计划了。当然，你计划的时间跨度越长，你做计划的水平就应该越高。一天或者一周的计划可以看成是战场上的短期战斗计划，让 30 天、60 天和 90 天的计划关注你想要实现的战略性目标上。

对软件开发师来说，思考自己在 90 天之内能完成什么工作有些强人所难。我们是有谋略的人。强迫自己想象 90 天后系统、团队进程，或自己工作的最终状态，这会使一些出乎意料的事情浮出水面。同环顾四周相比，鸟

瞰全局会带给我们完全不同的感受。一开始会很难，坚持住。就像所有的技术，熟能生巧，你和你的同事都会看到好的转变（即使你的同事不知道你的所作所为）。

状态报告可以帮助你推销自己。

Status reports can help you market yourself.

你应该与上司讨论你的计划。当你至少完成一轮计划之后，再开始讨论。重点是，在你的领导要求和你讨论计划之前，主动与他们交流。每周收到一封员工上周工作结果汇报和下周工作计划的邮件，不会引起经理的反感。经理们正希望员工能够主动发给他们这样的邮件。

一开始按周与上司进行交流。当你觉得适应了这个过程之后，开始按你的30天、60天和90天计划行事。更长远地想，计划出你想对项目或者你维护的系统做出什么高端有建设性的改进。把长期计划按提案的方式陈述给你的经理，并要求他们提供反馈。坚持一段时间之后，你会发现自己已经知道哪些事项经理不会提出异议，哪些是他们会提出较多异议的，经理对你的预期计划做出的调整也会越来越小。

做计划时要时刻谨记的是，出现在计划上的每一项工作必须要与后续工作相关，要么被完成、推迟、去除、或者被代替。如果计划上的某一项工作后来根本就无人问津，别人就会开始对你的计划持怀疑态度，计划也就失去了它的功效。就算最后的结果是不好的，你也应该同其他人交流。每个人都会犯错，但你应该坦诚地承认自己的错误，请求别人帮助来解决问题。坚持按照计划工作会给人留下一个积极的印象——在混乱中，不会遗漏任何一件重要的工作。

坚持这么做，有一天你的上司会发现你是可以掌握全局的。制定计划并按计划行事证明你不只是一个会编程的机器人，你是一个领导。裁员的时候，这样的人才不会被解雇，他们才是公司需要的人才。

与大家交流你的计划还有一个好处就是你做出的承诺会更具可信度。如果你不仅说出想要做什么，并且付诸实践完成了这项工作，你就会得到实干者的美名。有了信任就有了影响。如果你想要向某个机构介绍一种新的程序，比如是敏捷开发实践[①]，或者你想引进一项新的技术，有了信守诺言的信誉，尝试新事物的时候你就会被赋予更高的自主性。

在印度班加罗尔软件中心，我们有一个团队已经有一年多的时间负责夜班工作。这个团队共有七名成员，其中两个人一直都在上夜班。他们每周一轮班，每名成员每隔三、四个星期就会从晚上 7 点工作到凌晨 3 点。这个团队的成员心情开始郁闷，最终爆发了，抱怨自己一直都在倒时差。但是这个团队是重要的支持团队，美国国内的客户已经离不开他们实时提供帮助了。

后来，这个团队就制定了一个计划。他们研究了所提供的不同支持程序及相关方法，制定出一份计划，使所有的工作都在日班时间完成，同时显著提高客户服务质量。作为这一软件中心运行领导，我帮助他们对这份计划稍稍做出了一点改动（以此作为精神上的支持），将此计划以正式提案的形式提交给美国的经理。

他们的经理要直接与美国的客户沟通，所以这个问题将会是一个敏感的话题。会议开始的时候，团队成员自然会有些不安。但是，这个经理居然立刻就在提案上签字了，并且是非常愉快地。团队就把这份计划付诸实践，几个星期后，他们的时差消失了，每个人都回到了日班的工作。

这份计划的可靠性在于它不仅表述了要如何改变工作时间，而且也表明团队将在战略上提高工作表现，这点大大提高了上司和客户对他们的信任度。在就这种改变同客户进行沟通的时候，经理使用的就是这份计划，而团队也完成了既定计划。不出几个月，这个团队的工作效率得到了显著提高。他们得到了信任后，对工作更有归属感，工作也更具独立性。

---

① http://www.agilemanifesto.org。

碰到问题的时候，这个团队制定出一份计划作为解决问题的方法。他们不是向上司抱怨，而是做出了解决问题的提案。

你的上司希望你具有独立性和归属感。制作计划、执行计划以及与上司就计划进行沟通都会帮助你找到工作的独立性和归属感。

失败和模仿

——麻省理工大学学生 Patrick Collison

Larry Wall 曾写过："卓越的程序员的显著特征是懒惰、不耐烦和骄傲自大。"我不知道这些特征是与生俱来的，还是可以通过勤奋修炼学来的。无论是通过哪种方式，都没有明确地指明一个人真能利用这一信息来成为一名优秀的程序员。所以，我们应该注重有利于我们取得进步的行动，而不是特征。

如果让我来选择两种能使我们取得进步的方法，我会选择失败和模仿。

比起其他我所认识的程序员，我经历过的失败比他们要多。当然这就意味着我负责的大多数项目都失败了。参与的那些鬼项目不过是一次次徒劳地想做点儿有趣的事情，成功的机会其实很渺茫，就像热锅里煮着的龙虾，折腾来折腾去，终免不了死路一条。很有趣，就像托尔斯泰说的那样"幸福的家庭都是一样的，不幸的家庭各有各的不幸"。

尽管大家都说，经历过失败的公司对你而言就是不错的经验。但是就编程来说，我却没怎么听说过这个观点。

(这两点都是我擅长的，在经营上我也曾失败过。)

商业上的失败会产生出一种很直接的经验。你学到节约开支的重要性，或者你会变得更加坚定。但是对编程来说，从可能会失败的项目中学到的知识要更有价值。

我刚开始编程的时候，大部分时间都花在失败上了：编写操作系统、文件系统、虚拟计算机、重新实现网络通信协议、解释程序和运行时编译程序的编译器。这其中的大部分都没能正确运行，就算是能运行的，工作状况也不怎么样。当然，就算忽略技术层面的可行性，大多数工作从一开始就注定了要失败。我不知道仅有 1%的成功机率的新操作系统占有多大的比例，但是 1%的成功率确实很小。

对我来说，做这些项目的编程才是最享受的。这些问题才是软件工程中最基本的问题。这些工作都是围绕如何在空间、速度、可靠性和复杂性之间寻找折中点，无需解决软件升级过渡或修补 API 的漏洞问题。

就像我通常做的，可能你沉浸在这些问题中，潜心钻研了几个月，却始终找不到解决方法。

我发现现在学习编程的人已经不会再经历这些事情了，但我说不出确定的原因。

我认为至少有部分原因是因为互联网软件的兴起。就在几天前，骇客新闻网上还有人提问说现在到底还有没有人对编写客户端软件感兴趣。这么说有点夸张，但事实也大致如此。当然，互联网软件也是很棒的。

但是从编程的角度来看，这种转变还是会带来不利的影响。除非达到很广的范围，否则 Web 应用中很少包含严峻的技术挑战（不过 Internet Explorer 6 的兼容性问题却很多）。

换句话说，现在想要失败是很困难的，你必须要首先经历成功。

所以在这种情况下，我认为主动去寻找有失败倾向的项目是很重要的。

抄袭怎么样？大家都会说，要想成为更好的程序员，你应该阅读好的程序。尽管他们的意思可能不是字面上的意思（阅读程序有点太无聊了），“读”这个词好像不应该出现在这里，因为它听起来有些被动。但是我认为应该坦然地主动大范围地抄袭。

抄袭在很多事情上都适用。Hunter S. Thompson 不只是阅读好书，他也曾引用海明威和菲茨杰拉德的句子。人们所知道的巴赫最早期时候的手稿也是他对风琴手作品的改编。更著名的可能是比尔·盖茨，他在哈佛大学的垃圾桶里掏出一些程序设计师的笔记和字条，然后对着这些宝贵的资料研究操作系统。

抄写的功效是很容易就能被看出来的。抄写可以建立肌肉记忆。通过抄写你可以感觉原文的微妙之处和它的结构——如果只是粗略的的浏览，是无法发现这些细节的。

对编程来说，还有一个非常重要却不明显的好处。在处理那些看起来会失败的项目时，抄写可以让你走得更远。比如说一个散列表执行程序的副本（它让我写的第一个程序解释器没有那么糟糕）或者一个受到原作启迪影响的设计（例如 Linux 就是由 Minix 发展而来的）。

这就会产生失败和抄袭的良性循环，这个循环也是你评价自身进步的一种简单的方法。在处理棘手的项目时，你被一个不可逾越的难题绊倒了，你抄袭了别人的解决方法，结果是，不管问题是什么，现在你都知道该如何处理了。

这是一场无节制的掠夺，当你全心全意地去汲取各种各样的技术时，你会找到一种方法将这些技术以一种新的方式结合在一起。毕加索曾说过“好的艺术家会抄袭，而巨匠会偷”，我不知道这句话的真实含义是什么，但是前半句话的意思就是我一贯的主张。

编程的过程总是充满古怪的念头。使用较短且较少描述性的名字可以使代码更具可读性。最强大的语言通常都包含最少的概念。要想有成功的原创，失败和抄写可能是最佳途径。

第 4 章

推销……不仅仅是迎合

33 不要忽视感觉
34 探险向导
35 学会沟通，善于写作
36 到场
37 适当的言语
38 改变世界
39 让人们听到你的声音
40 创建自己的商标
41 发布你编写的程序
42 变为卓越的能力
43 建立关系



在你认识的软件开发人员里，你是最棒的。你的想法源源不断，设计十分高雅；在同事之中，你的架构洞察力无人能及；你编程的速度和精确度，也是首屈一指。

那又怎么样？

很多软件开发人员，特别是那些自以为是的开发人员，都认为自己的能力在同行和上司眼中是不证自明的。他们可以自然地将这个谎言掩藏于虚构的道德逻辑中：不显示自己的能力，是因为自己过于谦虚，炫耀自己的能力会像是阿谀奉承。有自尊心的程序员是不会在老板面前讨好卖乖的。

事实上，这都是他们恐惧的借口。

学校的各个团队挑选成员的时候，这类程序员在那个时候大多数都是最后一个被挑走的。只要可能，他们就会避免出现在一切社交场合，要是不得不去，表现也会一塌糊涂。所以，这些人不敢向别人展示自己的能力，也就没什么可奇怪的了。

先把怀疑放在一边，让我们暂且相信那个道德逻辑是真的。可不管目的是什么，隐藏你的能力都是非常愚蠢的。想一想：公司雇用你是来开发软件，为公司创造价值的。团队领导的工作就是发展团队，使这个团队最大程度地为公司创造价值。如果领导压根就不知道团队成员能够胜任什么工作，那他又怎么能开展工作呢？

不久前一个经理告诉我，如果一个人完成了一件非常漂亮的工作，却没有人知道，那在这个经理的眼中，这事情就等同于从未发生过。这听起来可能有些残酷，但是站在公司的角度上，这是有道理的。实际点来说，经理不可能对每一个员工每天的工作都掌握得一清二楚，而且不论是公司还是员工，也都不希望经理在这上面花费时间。公司希望经理可以关注大局，而不

是被琐事所纠缠。员工（特别是程序员）也讨厌被死死地管住。

简而言之，就算你的产品非常优秀，史无前例，但是如果你不宣传它，那就没人会去购买。众所周知——尤其是在软件这个行业里——最好的产品不一定就能在竞争中取得胜利。市场宣传更为重要。在职场中，也不要忘记这一金科玉律。

那应该要做些什么呢？

表面上看，宣传自己很简单。你的目标有两个：让别人知道你的存在，以及让他们知道，当他们碰到难题时，你是那个可以解决问题的人。这不仅适用于整个职场，也适用于你目前正在工作的公司。不要认为这个公司雇佣了你，管理者就一定知道你的存在。而且，就算你上司知道你的名字，他对你的能力也是一无所知。

本章不仅可以帮助你让你的上司知道你的能力，而且还会帮助你拓宽行业视角。本书中，我们已经讨论过如何使自己在职场上具有竞争力。接下来，我们要讨论的是如何付诸行动。

33 不要忽视感觉

扮演理想主义者的角色非常自在，你可以假装不在乎别人怎么看你，但，那是在游戏中，而你绝不能信以为真。你应该在乎别人的看法，别人对你的认识就是现实。好好解决它。

你可能听说过这个古老的心理学问题：“如果森林里的一棵大树倒下了，却没有人听到它倒下的声音，那么它倒下的时候到底有没有发出声响呢？”这个问题的正确答案是：“谁会在意这个？”

树倒下的时侯可能制造出了声响，但在抽象层面上，这不是一个振奋人心的答案。如果没人听到它倒下去的声音，那么树倒下去制造声响的这一事实其实是无关紧要的。

你的工作也是同样的道理。如果你非常出色，但并没有人知道，那你真的优秀吗？谁会在意？没有人会在意。

在印度 IT 官僚制度的亚文化中，人们居然连这么简单的事实都不明白。我接触过的人中，基本上没人知道让上司知道他们在做什么有什么意义。如果你知道你比哪个同事出色，那这点就该体现在你的工作表现评价、工作评分和工资中。但这些人却认为其他人会通过某种方式察觉到这个事实。

事实是什么？谁来给它下定义？绝对意义上的好与坏又是什么？

根本就没有绝对意义上的好与坏，至少在判断谁更具创造力，谁更有知识的时候，没有绝对的好与坏。怎样界定什么样的音乐是好音乐？什么样的图画又是好的图画？你可能会有自己的判断标准，但这都是你的主观判断。

绩效考核永远都不会是客观的。

Performance appraisals are never objective.

公司和人力资源部门都不愿意承担风险，所以他们企图用客观的评价方法来评价员工。有时候他们会使用“客观”的绩效考核系统。在印度，我的团队成员都认为这种绩效考核系统可以正确地评定他们的工作表现。但这是因为这些员工从没使用过这类系统。

事实上，对知识型工作者的能力以及他们的工作质量进行客观的评价是根本不可能的。还是不同意我的观点吗？那再想想你驳斥我的根据，看到漏洞了吧？

因此，既然说公司（或者整个行业，职场或者任何地方）对你的评价是主观的，那就意味着对你做出的评价总是基于别人对你的感觉。你升职加薪的可能，甚至是公司决定是否再继续雇用你的决定因素完全取决于别人对你的感觉。

主观上来讲，其他人依据个人喜好对你做出的评定，肯定不会出现两个相同的意见。不同的人有不同的喜好。有人喜欢严格的代码结构，有人喜欢松散自由的结构。有人喜欢通过邮件沟通，也有人喜欢电话沟通或者面对面地沟通。有些上司喜欢有上进心的雇员，也有的上司就希望员工表现得像个下级，上司说什么，就是什么。

这不能仅仅归结为个人喜好。人们的角色不同，与你的关系也不同，就会导致他们评价你时的侧重点也不同，而出发点是如何让你们的关系正常运作。如果我是项目经理，那比起你编写的源代码的质量，我会更看重你的沟通能力。如果我是你的同事，同为程序员，我会更看重你的天赋和创造力，而不是你跟进项目的能力。但是，如果我是你的上司，如果你没能做出什么成绩，那么对我来说你的天赋就没有半点意义。

很多人认为迎合别人的感觉是不体面的。但是，就像你所看到的，它是非常实用的。当你明确地了解其他人判断你的因素后，你就会更加明确如何可以让他们满意。对那些非技术层面的商业客户来说，他们不会对你面向对象的程序设计技巧留下深刻的印象。你可能是一个设计天才，但是如果你无法与他们进行良好地沟通，也无法按时完成工作，客户就会认为你非常差劲。这不是他们的错误，而是你的确差劲。

感觉非常重要。公司是否继续雇用你，提升你或者让你常年留在同一岗位上，加薪还是减薪，这些都会受到别人对你的感觉的影响。你越早明白这个道理，并且能够控制别人对你的感觉，那你就能越快回到正轨上。

练　习

影响感觉的因素因人而异。你的母亲不会在意你面向对象的程序设计技术怎么样，但你的团队伙伴会在意。

在人际交往关系中，你要弄清楚哪一因素对哪一种关系圈是重要的，这样你就可以给你周围的人留下可靠的感觉。想一想办公室里与你有着不同关系的人。比如，与你做着相同工作的同事，你的上司，一个或者几个客户，还有项目经理。

把他们分组（不管你办公室的关系结构是什么），并罗列出来。在每一项旁边记录下你的哪些特质会影响他们对你的感觉。下表为例。

表4-1　不同关系组所需的特质

组	影响其感觉的特质
团队成员	技术水平、社交能力、团队精神
经理	领导能力、客户服务意识、沟通能力、项目跟进能力、团队精神
客户	客户服务意识、沟通能力、项目跟进能力
项目经理	沟通能力、项目跟进能力、效率、技术水平

依据实际情况，更换列表中的内容。根据这个表格，在工作中你会做出哪些改变。你已经做了哪些改变，而哪些改变是你还没有做的？

34 探险向导

在公司里，让人们了解到你的沟通能力是非常重要的。程序员蓬头垢面，借助显示器的一点亮光，在终端机上苦战的日子已经不复存在了。

这一观点可能有些令人烦恼，但是请你站在经理和客户的立场上来思考问题（在本节中，我将用客户一词来代替经理和客户）。

他们负责一些至关重要的事情，这些事情最终还是要委托给那些可怕的IT员工来执行。他们竭尽所能地推进项目的进展，但最终还是得任由程序员摆布。而且，他们也不知道如何来控制这些程序员，甚至无法通过交流来搞清楚他们到底在做什么。这种情况下，他们希望团队成员具备什么特质呢？我敢打赌，肯定不是希望程序员能记住最新的设计范例，也不是程序员知道多少种编程语言。

他们寻找的是可以帮助他们完成项目的人。

> 客户害怕你。
> Your customers are afraid of you.

我们所说的这些经理和客户有一个小秘密——他们害怕你。原因是你很聪明。你使用的神秘的语言，他们不懂。有时候你的评论会很讽刺，这让他们觉得自己非常愚蠢（而你自己可能都没意识到自己的评论很讽刺）。你的工作往往是使得一个项目构思变成事实的最后、也是最重要的一步。

你的工作就是做客户的IT向导，带领他们穿越IT世界里那些陌生的地带。当客户通过陌生地带的时候，因为有你，他们会感到舒服自在。你向他们介绍沿途风景，带他们去想去的地方，并且根据自己以往的经验，避开那些破旧的地方。

一般来讲，非程序员和程序员一样聪明（也就是说，他们中的大都数都不是很聪明，但有一小部分人非常聪明）。所以，你的客户很可能和你一样聪明，只不过对编程一窍不通。这不要紧。对他们的日常工作，你一样也不懂。这就是你们同时存在的原因，也是为什么公司要雇用你们两个人来一起工作。

我之所以提到“聪明”这个问题，是因为计算机界人士经常认为不会操作电脑的人都是不聪明的。这样直白地说出来，感觉非常傻，但这就是偏见。这种观念在很多人心里根深蒂固，自己却毫不知情。刻意地控制这种感觉是没用的。

我的建议是调换角色。不要感觉自己是一个计算机天才，从计算机天堂降临，来拯救那些身处炼狱中的可怜客户，扭转一下这个局面。打个比方，如果你在保险行业工作，把你的客户想象成保险这个行业中某一课题的专家，为了完成你的工作，你必须要去学习那个课题。

这样说来，你应该知道当你在谈论关于软件的问题时，应该降低一点难度，来满足客户的认知。在过于专业和过于愚蠢之间可以找到一个微妙的平衡点。

“为什么现在讨论的都是如何对待客户？不是要讨论怎样推销自己吗？”如果你在一家典型的 IT 机构工作，那么之所以有预算来使你领薪受雇，那都是为了实现一项业务职能——你客户的工作也是为了实现这项职能，他们所做的决策也是依据它。决定升职和人员安排的时候，你最有力的支持者就是一个无法离开你的客户。反之，想象一下如果你的客户认为你总是居高临下，后果又是什么？你的客户代表业务的需求，不要忘记你正是因为他们的这些需求而被雇用的。

练 习

(1) 自我检查——你是不是一个令人生畏的编程老恶魔？你确定么？他们是不是害怕告诉你真相？检查你的邮箱，找到一些你发给不太具有编程专业知识的同事、经理和客户的邮件。尝试站在他们的立场上再次阅读这些邮件。如果距离这些邮件发出已经有一些时间了，你会发现自己可以以第三方的角度来看问题了。

还有一个更好的方法，把这些邮件给你母亲看。告诉她这是你的一位同事发给客户的邮件，问问你母亲对这些邮件的看法。

(2) 跳过围墙——寻找一个自己知之甚少、需要依靠别人的境况。

如果你脚步动作极其笨拙，就想象自己是一名足球队员。如果你手部动作极其笨拙，就想象自己是国家编织队的一员。这种情况下，你会希望队友如何与你沟通？

35 学会沟通，善于写作

程序员的沉默时代已经结束了。如果公司要想制造沟通障碍，那他们就会把程序员安置在另一块大陆上，另一个时区里，然后只通过邮件和电话与他们联系。

文字表达能力是非常重要的。可是，在所有你为了保住职位而需要锻炼的技能中，这一条听起来有些做作、傻，或者微不足道。你可能会觉得有点像回到了高中的英语课上，但这次你必须要专心致志。

我们首先解决最枯燥的部分：语法和拼写。你可能已经拿到了工程学或者计算机技术的学位，而我却还在这里不知天高地厚地告诉你学习如何拼写。我可真是胆大包天！

但是事实上，这个问题是存在的，至少存在于美国社会中。

根据美国国家写作委员会的一项报告指明，参与此次调查的公司中，一半以上公司在做出雇用和升职决定时，会将写作技能作为决定因素之一；参与调查的公司中 40%为服务业，这些公司表明在新进员工中，只有 1/3 或者更少的员工具备他们所需要的写作能力。[①]

退一步看，纵观全局，写作能力不但非常必要，并且还很紧缺。

我们都知道，劳动力正在全球范围内自我分配。在这一大趋势下，一个时代即将来临——对有些人来说，这一时代已经来临！——这就是工作沟通将借助文字形式，无论是通过即时消息还是邮件。

---

① http://www.writingcommission.org/report.html。

这种情况下，如果你的工作中要包括很多写作内容，那你最好要擅长写作。你的写作能力将会成为人们对你做出评价的依据。或许你是一名出色的程序员，但如果你不能很好地通过文字表达自己，那么在一个有空间距离的团队中，你就无法有效率地工作。

人们会通过你的写作能力对你做出初步评价，也可以依此深刻理解你的思维活动。如果你无法用母语清楚地表达自己的想法，让别人明白，又怎么可能用编程语言来清楚地表达出来呢？组织观点，带领读者思考并最终做出合乎逻辑的推断，这种能力与创作出清晰的设计和系统实施，并让功能维护者理解的能力是一样的。

写作能力不仅仅是判断的依据。如果你的团队成员和你处在不同的时区内，那么写作可能就是你汇报工作进度的唯一方法，比如你是如何设计某种东西，或者你的团队成员需要做的工作是什么。

你自己就是你要解释的内容。
You are what you can explain.

沟通，特别是以文字方式进行沟通，是你所有绝妙的想法必须要通过的瓶颈。你自己就是你要解释的内容。

练 习

(1) 开始记录开发日记。每天写一点，记录你做了什么工作，解释你的设计决定，检查棘手的技术和专业决策。即使你自己是第一位读者（或者是唯一的读者——这由你自己决定），也要注意写作的质量，和能够清楚表达想法的能力。时不时地回头阅读之前的日记，评论它们。通过你对之前日记的喜好，来调整你的新记录。这样做，不仅可以提高你的写作能力，通过这些日记你还可以加强你对所做决定的理解，当需要知道如何或者为什么你之前要做某事的时候，你就可以在日记里找到答案。

(2) 学习打字。如果你不是按指法打字，那就去参加一个课程或者下载一个学习软件。在习惯了输入技巧后，在写作的时候你就会更加舒服和自然。当然，如果打字速度快，那么在写作的时候也会节省很多时间。

36 到场

能够与上司和客户面对面地沟通是你的优势，不要浪费这个机会。

我在印度班加罗尔做软件开发中心 CTO 时，向经理报告工作的经历很不愉快，我不喜欢那个经理（她也不喜欢我），她人在美国。我们只能在深夜或者清晨通过电话沟通，背景噪声和电话突然断线使沟通的过程非常令人沮丧。为了拉近距离，解决时差问题，我给她撰写了很长的邮件，但却得不到她的回复。如果我向她抱怨我的邮件被她忽略了，她就会批评我写的邮件太长。我无法解决我们之间的问题。

那时候公司有年度绩效考核，在考核中经理要列出雇员的优点和（所谓的）发展需求。那年我的发展需求的第一项就是到场。

本文所指的到场是一个绝对的公司用语，它表示一种非常模糊的领导特征。它包含一种无法估量的才能，即让别人感受到你的确在场，尤其是在面对面的情况下。它还包含另一种无法估量的才能，即让自己的举止看上去像个领导。

我坐下来通过电话与我敬爱的经理谈论绩效考核时候，当她说道“到场”的时候我盖住了话筒，因为我不想让她听到我的笑声。我怀疑在剩下的对话中，她有没有感觉出来我是做着鬼脸，半微笑地说话。我们俩都知道真正的问题就是“到场”这个词的基本含义：我只不过没有和其他人一样在美国。

我的同事中大部人都不喜欢这个经理。她很少做值得尊重的事情，所以这是意料之中的事。按照上述模式，与她关系紧张的员工都是和她不在一个地方工作的人。

那些在印度、匈牙利和英国（按照降序排列）工作的人和她的关系都很紧张，因为我们不仅和她不在一个地方，而且时区不同，行政机构、文化和语言的不同也都是导致沟通不畅的因素。

可是，即使那些在美国工作的人也在竭尽全力地躲避这个经理。近距离接触和偶尔面对面地交谈才会使这名经理感到舒服。当然，我刚一到印度，就体会到“眼不见，心不烦”了。

听我讲述了这个故事后，你可以从中学到一些东西。在工作中，近距离的接触是一种优势。

回想最近一次不懂电脑的亲戚朋友向你请教计算机问题时的情景。你试图通过电话帮助他们解决问题，但是如果他们听不明白，你的情绪就会越来越激动。你会想我要是能向他们展示怎么做就好了。相比之下，面对面地交流就会非常地有效率。你可以更加清楚地掌握对方的问题，通过手势和画图这些视觉方法来解决问题。你还会不自觉地使用面部表情来表达要说的内容。

通过面对面地互动，我们不仅提高了工作效率，增进了沟通，还形成了更加紧密的人际关系。如果你没有见过某人，要想建立友谊就要花费很长时间。现在，互联网无处不在，不见面就能建立友谊也是比较普遍的，但在15 年前，这是不可能的。但通过电话、邮件和聊天工作效率比较低，通过这些方式建立人际关系的效率也不高。而且这种通过邮件和聊天软件的对话也不是很舒服（下一代人可能不会有这种感觉）。大多数情况下，这种远程工作环境中建立的关系都是以完成任务为中心的。

通过有效、高带宽的交流建立的团队关系可以更快更好地生产软件。在大多数环境中，重要的项目决定都是在喝杯咖啡的时间和在闲聊的时候做出的。如果你是其中一员，优势也是显而易见的。当然对我们来说，最重要的是能够被别人看到。

我从来都不去银行，都是通过网络或者自动存取款机办理业务。我爷爷奶奶就不同。他们会到银行柜台去办理一切业务，甚至也不喜欢通过电话办理业务，这样做会让他们感到很不舒服。他们和常去的小商店的工作人员也成了朋友。他们一次又一次地光顾，在结账的时候与他们交谈。他们不会考虑去别的小商店（或者银行），因为他们选择银行或者小商店的原因不是务实地考虑成本和方便，是因为人。

在由机器人和计算机程序来评判我们的绩效成绩之前，所有业务都是通过人来进行。人类喜欢与人打交道，至少大部分人是这样的。

计算机工作人员自然的工作模式是藏在一间小屋或者办公室里，戴上耳机，开始埋头苦干，一直到吃饭时间。Douglas Coupland 在 *Microserfs* [Cou96] 一书中讲述了一个笑话，说一个程序员在办公室里做项目，团队必须要为他买形状是扁平的食物，因为只有这样的食物才能被塞进门缝，滑到程序员那里。这种隔离的状态，已经成为软件行业的一种文化和风俗了。

然而，这样对你的职业发展没有什么好处。如果你被锁在办公室里，只通过电话（如果你接听）和邮件与外界沟通，或者还要工作到深夜，那这种工作状态和你被派到国外工作就没有什么区别了。你错过了在公司中成为骨干的好机会。记住，要与人打交道。你必须要记住人类的天性是喜欢与人一起工作——不是和语音信息、邮件和即时消息一起工作。

了解你的同事。
Learn about your colleagues.

在当今的工作环境中，尽管你和工作伙伴在同一个国家工作，却不一定在同一个城市或者同一个州。这种情况下，如果有可能，那么就应该定期出差与他们见面。不过最好的办法还是打电话。打电话给你的老板和同事的时候，要拿起听筒，而不是使用扬声器。也不要依赖定期会议。你需要去模仿那种在喝咖啡的休息时间进行的自然交谈，就像你和他们在同一个地方工作的时候一样，所以给即兴对话留出时间。时不时地，营造出私人对话的感觉。以“今

天过得怎么样？”开头，接着问“你周末通常都做什么？”尝试去了解你的同事。这样做，不仅可以牢固你在公司的地位，也会让生活变得更加丰富多彩。

练 习

(1) 下周的某一天，强迫自己不发邮件（在合理范围内）。当你想要发邮件的时候，打电话给对方或者到他的办公室面对面地与他交谈。

(2) 想想你与哪些同事、上司和顾客的交流不够，列出名单。在日历上标注出给他们打电话或者与他们联系的时间（通过电话或者面对面）。谈话要简短且有意义。与他们谈论工作或者只是单纯进行人际沟通。

37 适当的言语

我的小侄子们经常使用电脑。相对来讲，他们对电脑还是挺在行的。他们用电脑与世界各地的朋友联系交流。他们喜欢使用聊天工具、电子邮件和浏览网页，当然也使用所有高中生在做功课时都喜欢用的一些功能。

但如果我对他们吹牛说我的新电脑硬盘为每分钟 1 万转速的转串行 ATA 硬盘，他们的反应至多就是非常幼稚地佯装出非常热情的样子。如果我告诉他们这台电脑有数千兆字节的随机存储器，它的 GPU 速度也比我 5 年前用的 CPU 还要快很多，他们可能也不会有什么特别的反应。

但是，如果我说用这台电脑全屏玩最新第一人称射击游戏时，画面非常清晰。听到这句话，他们会非常感兴趣。

14 岁大的男孩对电脑游戏有浓厚的兴趣，但是千兆赫兹和每分钟的转数对他们没什么吸引力。

商务人士对转数和千兆存储容量也不感兴趣。他们在意的是应用程序的速度够不够快，却不管新的定制应用程序服务处理器每秒钟能处理多少请求。

请用行业术语推销你的成就。
Market your accomplishments in the language of your business.

商业本身关注的是结果，经营者关注的也是结果。所以，如果不使用行业语言来推销你的成就是起不到作用的。

你不会用德语向美国观众推销产品。饮料公司不会以饮料中色素的测定量作为卖点。众所周知，推销产品时，要使用消费者明白并且与他们相关的语言。

也就是说，作为软件开发师，要把你所完成的工作放在你所服务的行业的框架里。你确实做了某些工作，但是这工作是什么？这项工作有什么用？怎样才能证明你所谓的成绩不是在浪费公司的时间呢？

我猜，如果回想上个月你做的工作成绩，你可能都说不出来这些工作为何有用。有人把这些工作分配给你做，但对公司而言，这些工作的意义又是什么呢？

在通用公司有这么一个传说，前执行总裁 Jack Welch 喜欢与随意碰到的通用员工一同乘坐公司某一座高层大厦的电梯。与他一起乘坐电梯的员工早已惴惴不安，但是 Jack Welch 还是会转身问这名员工："最近在做什么工作？"然后会接着问："这项工作的意义是什么？"（这才是真正刺痛员工的问题）。这个故事的真谛在于，它告诉大家要对电梯演讲做好准备，以防万一。

如果你公司的执行总裁突然问你这个问题，你会怎样做出回答？即使给你几分钟时间做准备，你能够解释出你正在或者最近刚刚完成的工作的商业意义是什么吗？对于一个完全不懂技术的高级执行总裁来说，他能够听懂你的答案，并对其表示赞赏吗？

练 习

(1) 罗列出你近期完成的工作，并写出每项工作的商业意义。如果对某项工作的商业意义不甚了解，可以请教你的上司或者某个你能够信任的熟人。

(2) 准备好你的"电梯演讲"，并将其牢牢记在心里。

38 改变世界

对你而言，工作中最不好的事情就是有人问“他是做什么的？”这就意味着别人根本不知道你到底做了什么工作。

这有些悲哀，但是我确实不知道很多在大公司里与我一起工作的人都做了什么。这些人去上班，完成分配的任务，然后下班回家。他们留下的除了一连串的代码、文档和邮件，没有什么持久的影响。

这种状态就是没有带任何任务去上班。你只不过是等待着别人告诉你要做什么工作。当你做着别人分配给你的工作时，唯一知道你在做什么的人就是分配给你工作的人。如果你在零售业工作，或者你只想做一名外包团队程序员，这样也无大碍。

> 带着任务去上班，并确保别人知道你的任务。
>
> *Have a mission. Make sure people know it.*

但是如果你想在高消费城市里做一名软件开发师，你就必须要带着任务来上班。你必须要做出变化，而这种改变不只是你自身或者是你自己工作的改变（这只是前提）。你所做的改变必须要让你的团队、组织或者公司看得到。

这种改变可以是一个小变化。做单元测试时，你可以向公司里那些新手程序员展示你所做的测试。这种改变也可以是一个大变化，比如介绍一种可以节省开支，却能使系统运行更快更好的全新技术。

你做这些事情是因为内心里觉得必须要这么做。你无法看着公司里的同事犯错误，却假装视而不见。你知道有更好的办法，你必须要改变现状。

当然，你做出这样改变肯定会引起某些人的不满。但只要你的意图是正

确的，那就没关系。这种改变不要太突然太猛烈，但也不要总是小心翼翼、畏手畏脚。

如果最终你还是惹毛了某些人，那至少他们不会再问“他是做什么的”这个问题了，这不是很好吗？

如果你不知道要做的改变是什么，那你就没有在做任何改变。如果你没有主动让自己获得承认，那你就还没有获得承认。

练 习

记录下工作中你亲眼看到过的改变。想想哪些同事是带着任务在工作，哪些同事是最努力最有效率的。他们的任务是什么？

你能想到这些任务中哪些是不合适的吗？努力和狂热的界限是什么？你的同事中有没有人越过了这条线呢？

39 让人们听到你的声音

目前为止，我们所探讨的观点都是非常保守的，主要集中讨论如何让你的工作成绩被认可这个问题。不管是你想赢得别人的注意，取得进步，还是想保住饭碗，这个问题都是十分重要的。

但是，这些内容实在枯燥乏味！

世界在变化。要想做出成绩，就必须要比去年的 IT 工作人员想得更加长远。或许你的近期职业目标是从 23 级程序员升到 24 级，你要思考的应该比下一次升职长远，甚至不能只考虑在你当前公司里的发展。

把目光放得更远一些。不要把自己局限在某一特定公司中的程序员——毕竟，你不太可能永远在一个公司里工作——因此，要把自己当做是某一个行业的人员。你是一名手工业者或者是一名艺术家。除了你为公司人力资源部开发的开销报告系统，或者是你发现的公司问题追踪系统中的漏洞之外，你还有更多可以与人分享的东西。

公司都想要雇用专家。列出一长串项目经历的简历当然是展示自己工作经验的好办法；不过，如果在面试前，面试官已经听说过你的名字了，那可没有比这更好的了。要是面试官读过你发表的文章或者撰写的图书，或者他们听过你的演讲，那就棒极了。如果你正想要开发某种技术，你难道不想雇用一个撰写过此类技术和方法的相关书籍的专家吗？

以前做专业萨克斯风演奏者的时候，我常在孟非斯比尔街上或者这附近的酒吧里演奏。当我开始转行到计算机行业的时候，我发现在音乐界出名的方法和在计算机这行出名的方法有很多相似之处。作为乐器演奏者，要想找到工作，就要记住以下几点。

- 最优秀的萨克斯风手不一定总能找到工作（这条是最重要的）。
- 和谁一起演奏是非常重要的，至少和你的演奏水平差不多；作为演奏者，联合的力量是强大的。
- 有时候，优秀的演奏者常常被忽略，因为人们总是觉得他们没有时间，或者是人们根本不敢去询问这些优秀的演奏者有没有时间。
- 人际关系网络非常重要。如果你并不认识某个乐手，那你就不太可能被邀请与此人一起演奏，除非有中间人介绍。

计算机这行也是一样。专门用来评估和雇用软件工程师的系统是不存在的。当然，优秀是非常重要的，但是只是优秀是不够的。我们这个行业，和音乐界一样，都是由一个复杂且广大的人际关系网构成的。你认识的人越多，得到好工作的机会就越大。如果只把自己局限在现在工作的公司里，就会严重限制你形成新的人际关系网的机会。

还有什么比出版文章和公众演讲更好的方法让人们听到你的声音、记住你的名字呢？那么，如何才能从一名程序员到出版作品再到公众演讲呢？从互联网开始。

第一步就是阅读网络日志。如果你不知道读什么日志，那就挑选出几位你最喜欢的技术类书籍作者，然后通过网络搜素，通常他们都会有自己的网络日志。订阅这些日志的，以及这个网页上链接的其他人的日志链接。慢慢地，随着你阅读和寻找其他人的网络日志链接，你的日志链接列表就会不断增加。

接下来，开始撰写自己的网络日志。有许多免费开办日志的服务。操作起来非常简单。一开始，你可撰写（或者链接）在你的聚合器模块中有趣的故事。慢慢你就会发现，网络日志这个环境本身就是一个社交网络——你开始建立的职业网络的缩影。你的想法会出现在别人的日志链接中，他们会撰写关于你的想法的文章，然后传播你的观点。

网络日志是训练场。要抱着为你最爱的杂志写专栏文章的态度，撰写网络日志。随着你写作技巧的增长，你也会越来越有自信。

你也可以使用网络日志上的内容作为下一步工作的范例。既然你已经在自己的论坛上撰写文章了，不妨将这些文章在社区网站、杂志甚至书籍上发表。这些在网站上的作品就是你写作能力的证明，也为你撰写文章或者出版图书提供了材料。文章成功发表后，你的人际关系网也会随之扩展。写得越多，得到的出版机会也就更多，而这些又为公众演讲奠定了良好的基础。

就像在网站上练习写作一样简单，你可以在当地开发员小组会议上开始你的演讲生涯。如果你是.NET 开发员，就为当地微软开发小组准备一次演示。如果你是 Linux 程序员，就在你所在的 Linux 用户小组中做一次演说。熟能生巧，不要小视这些演说，一定要在其中放入你的思想。尽管你演讲的对象只是你所在的城市中的一小部分人，但这是你生活和工作的地方。成功的演说一定会带来收获。如果你给予这些小范围的演说足够的重视，那你就会发现做这些小演讲的收获并不亚于那些在重要行业会议上的演讲，也就是你的下一步。

所有这些让人们听到你的声音的方法中，最重要的一点就是要尽早行动，而不是一味地去想自己是否已经做好了准备。大部分人都会低估自己的能力。你一定有某种可以传授给他人的东西。你永远也不会觉得自己已经做好了 100%的准备，所以不妨现在就开始行动。

练　习

如果你还没有网络日志，现在就创建一个。

在你的电脑里创建一个新的文本文件，列出一切可能的网络日志话题，这些就是你要撰写的专题文章。不要局限在宏大的观点上。尝试那些 10 到

20 分钟内就可以写出相关文章的小想法。当这个列表达到 10 项的时候，就停止（如果你的灵感停不下来，那就继续）。

保存文件，但是仍然保持打开的状态。如果你重新开机了，那就再打开这个文档。为自己设定三个星期的期限。每天，从这个列表中找一个话题来撰写文章。不要过多的思考，就是写一遍关于这个观点的文章，然后在网络日志中发表。在文章中加入其他网络日志中相关文章的链接。每天挑选话题的时候，可以任意向此列表中增加新的想法。

三周后，挑选出你最喜欢的两篇文章，提交到类似 Digg 和 Reddit 这类由用户审查文章的网站上。如果你的列表中还有未撰写成文章的想法，那就继续写。

40 创建自己的商标

创建商标分为两部分：认知和尊重。创立自己的商标，让人们认识它，然后确保与之关联的都是积极的特征。

现在，当我们看到“卐”这个标志，就会想到希特勒和德国纳粹。从确立商标的角度来看，这对纳粹来说是非常有利的。他们成功实现了创建商标的第一部分：认知。但是，对于思想正常的人来说，看到这个标志，我们想到的是大屠杀，即与之关联的信息是极度负面的。所以，纳粹最终在积极的关联信息这一步上惨败。事实上，“卐”是希特勒从印度教那里偷来的。在印度教中，“卐”是象征着吉祥和幸福。但是，在当今的西方世界中，这一神圣符号的名誉全毁。这就是得到广泛的认知，却没有得到尊重。

Charlie Wood[1]是一个相反的例子。Charlie 是一位不可思议的歌手、歌曲创作者以及在田纳西州孟菲斯的 Hammond B3 管风琴演奏者。每周，他在比尔大街的一家酒吧里演奏五个晚上。每个认识或者听说过他的人，都知道他是多么的令人惊叹，对他十分敬仰。他对蓝调布鲁斯极富天赋。

但是，相对而言，却很少有人知道他是谁。没有认知，却得到了尊重。

你既想得到认知，又想得到尊重。你的名字就是你的商标。

你的名字就是你的商标。
Your name is your brand.

本节所讨论的内容就是如何既得到认知又得到尊重。这里，你要懂得这二者的结合是值得去创建和维护的资产。那些大的吓人的公司的市场部常指责大学的孩子们在网上盗用他们公司的标志或者宣传语，你和他们不同，不需要时时提防别人盗用你的商标而花大量的时间进行保护。唯一潜在的破坏性因素就是你自己。

---

① http://www.charliewood.us。

不要小视你所代表的内容。小心挑选你名字出现的地方。不要去做糟糕的项目，广泛地发送糟糕的邮件（或者撰写糟糕的网络日志）。别做傻瓜。没人喜欢傻瓜，尽管有时候他们自己就像个傻瓜。

> **Google 永远不会忘记。**
> **Google never forgets.**

更重要的是，不要忘记，你选择去做的事情会对你的名字产生持久的影响。既然我们现在很多互动都是通过网络上的公共论坛、网页和邮件列表来进行，我们的行为都会被永久地公开记录、缓存、编入索引并可被别人搜索到。

有一天你可能会忘记自己做过什么，但是 Google 却记得一清二楚。

竭尽所能捍卫你的商标。别让自己的行为毁了它的名誉。你的商标就是你的一切。

练 习

用 Google 搜索自己——用 Google 搜索自己的名字。阅读前 4 页的搜索结果（居然真有 4 页？）。根据这四页的搜索结果，别人会推断你是个怎样的人呢？这些结果是否能看出你是怎样的一个人？你对这个结果满意吗？

再搜索一次，但这次请注意论坛和邮件列表中的对话。你是不是一个傻瓜？

41 发布你编写的程序

如果已经有公司在使用你开发的软件了，那么找工作将是多么容易的一件事啊！你会说："你在使用 Nifty++吗？我能帮助你，因为这是我编写的。"这样，面试的结果就会完全不同了。面试者和经理会记住你——正如你所希望的。

这个想法听起来非常不错，但是就在十年前，这种机会却是少之又少。你必须要首先为商业软件客户工作，而人们对你的信任也紧紧捆绑在你为客户开发的软件中。但现在情况不同了，你再也不必为那些大公司开发流行、名牌软件了。

现在又多了一条出路：开源。开源软件已经成了主流。随着 IT 行业开始了新的项目，长期以来的争论已经从"创建还是购买"转向了"创建、购买还是下载"。只要不是完整的应用程序，从小程序库到成熟的应用程序容器都可通过开源许可发布，并且成为实际的标准。

大部分开发此类软件的人，都是和你一样的人。他们晚上和周末坐在家里，由于热爱而不知疲惫地敲打着键盘。当然，也有一些公司资助的开源产品。但绝大多数的开源产品都是独立的开发师出于爱好开发出来的。

> 人人都能使用 Rails，但很少有人能开发出 Rails。
>
> Anyone can use Rails. Few can say Rails contributor.

尽管很多开发员这样做只是出于兴趣和享受过程，但诱惑因素还是存在的。他们在某个团体中扩展自己的社交圈；在为自己树立名号；在此行业中建立自己的名誉。或许这些都不是刻意为之，但是在这个过程中，他们将自己推向了市场。

先不说你可以为自己树立名号，对开源软件有所建树也展现了你对这个

行业的热情。即使某个公司没有使用过或者听说过你开发的软件，但这个软件由你创作并已经发表的事实本身，就是雇用时候的一个优势因素。这样想，如果你想要雇用一名软件开发员，候选者中其中一名开发员每天朝九晚五地工作，下班回家就看电视；另一名开发员非常热爱软件开发，他自觉地加班工作，周末也还在开发软件。这两名开发员，你会更青睐于哪一名？

对开源软件有所建树是一种技术的展示。比起在简历上空谈你知道某种技术来说，如果你正在编写真正的代码，并效力于某一真实的项目，那你就会更有竞争力。任何人都可以在简历上的技能一栏列出 Rails 或者 Nant。但极少人有资格写出他们也是 Rails 开发者或者 Nant 提交者。

领导一个开源项目展现出的能力远不止技术能力。它需要你具备领导才能，版本管理、编制文件以及支持产品和团体的技巧，这样围绕你的努力来形成一个团体。如果你在业余时间成功完成了这些工作——作为一个爱好——你就与其他应聘者大大不同了。大部分公司都不会付给开发员薪水来做所有这些工作。而你不仅能够胜任这些工作，而且就算没有报酬，你对这些工作也是非常地在意，这就展示出你非常具有主动性。

如果你创作的产品非常实用，那你就很可能会成功。或许在一个小的技术圈子成名——可能是在使用 Rails 的开发员中。如果你足够幸运，圈外的人或许也会听到你的大名，比如 Linus Torvalds 或者 Linus。就算你没有一举成名，发布你编写的程序也会让更多的人知道你。开源社区是全球性的人际关系网络，这些人搜索程序的时候，可能刚好就搜到了你的软件，然后安装了它，并开始使用。这样，他们就会知道你的存在，随着越来越多的人使用你开发的软件，你的名字和名声也会随之传播开来。这就是市场化。这就是你所期待的。

练 习

Stuart Halloway[①]在做了一个研讨会，他称之为“Refactotum”。如果你有机会参与，我极力推荐，要点如下：选一个带有单元测试的开源软件。在代码覆盖分析器中进行单元测试。找到这个系统中最少被测试到的部分，并编写测试来提高代码的覆盖面。未经测试的代码往往是无法测试的。通过重构可以增强代码的可测性。将你所做的改变作为补丁提交。

这样做的好处是它起到的作用是可以衡量出来的，并且可以在短时间内完成。所以你没有理由不去尝试。

① http://thinkrelevance.com。

42 变为卓越的能力

传统上市场营销的四要素是：生产、价格、宣传和定位。这一观点是说如果你全部掌握这四部分内容，就会制定出完整的市场计划。这四部分内容同等重要。

但市场的目的是什么呢？目的就是在生产者和消费者中间建立起关于某种产品或服务的联系。而这一联系的起点就是对该产品的认知。传统建立某种商品认知的方法是通过宣传，包括广告、邮寄目录和教育研讨会。

最近，人们又开始关注“病毒营销”。当某个宣传概念非常出色，使商品在消费者中一传十、十传百地散播开来，这就是病毒营销。它就像病毒一样扩散，每一个被感染的消费者都有可能将病毒传播给更多的人。

病毒营销之所以受到青睐，不只是因为它避免了邮寄目录和购买电视广告所带来的高昂费用。还因为比起电视广告和邮寄目录，消费者更相信身边的朋友。他们倾向于相信同事对某种产品的介绍，而不是夹杂在报纸中的宣传小册子。

营销大师 Seth Godin 在 *Purple Cow* [God03]一书中十分肯定地宣称要想让消费者对产品做出评论，最好的方法就是将你的产品做得卓越。Godin 称传统的营销要素已经过时了，消费者对老套的大众营销战略，即先广撒网，然后再祈祷的模式已经麻木了。他说，要想在人群中独占鳌头，唯一的方法就是确实卓越。

现在，挑剔的读者开始鼓掌欢呼了。你试图尝试的所有迷惑人的做法都比不上真正卓越的能力。别急着说“我早就这么说过”，我们先来探讨一下卓越的定义。

出色和好的含义肯定不同。通常，卓越的产品一定是好的。但是，好的产品却很少是卓越的。卓越的意思是值得被关注。仅仅比你认识的软件开发员好，并不能使你成为卓越的软件开发师。仅在量上超过其他人，是不足以让你的声誉像病毒一样扩散开来的。如果要求某人谈谈对你的看法，或许他会说出很多你的丰功伟绩，但是卓越的意思是人们会主动地谈论你，而不是被动要求。

要想卓越，就必须和周围的人大相径庭。大部分在本章中讨论的自我营销策略都是为了配合卓越。发布成功的开源软件、写书和撰写文章以及在研讨会上演讲都可能会提高你变得卓越的能力。

展示或者让它死亡！
Demo or die!

如果你再看一遍上一段中的最后一句话，你就会发现我所提到的使你变得更加卓越的几点方法中，都是要去“做”某事。你可能是最聪明的或者最快的，但是只具有某种特质是不够的，必须要做点什么。

Godin以“紫色的牛”来提醒我们如何才能变得卓越。他没有使用最好的奶牛、产奶量最多的奶牛或者最美丽的奶牛，因为无论是在最好的奶牛群中、还是产奶量最多的奶牛群中，亦或是最美丽的奶牛群中，一头紫色的奶牛都会脱颖而出，成为人们谈论的对象。

怎样做才能使你自己和你的工作成绩变得像那头紫色的奶牛一样引人注意呢？对一门科目而言，不要仅仅是掌握它——可以撰写一本关于此科目的书籍。可以编写一个代码生成器，使以往需要一个星期才能完成的工作缩短为5分钟。不仅要得到同事的尊重，还要通过研讨小组或者专题讨论会使自己成为城市中公认的某一种技术的专家。把以前从未有过的想法放到下一个项目中。

仅仅做人群中最好的是不够的。要成为人们谈论的焦点。

练 习

从小事做起，但是要在你现在的项目或者工作中做一些卓越的事情。比如可以力争卓越的效率。项目进程往往会有一些拖沓的部分。从中找到其他人认为要一个星期才能完成的工作，你用一天的时间来做完它，如果有需要就加班。不需要把加班作为一个习惯，但现在是一个试验。用非常短的时间来完成这项工作，看看其他同事会不会对此作出评论。如果他们没有评论，那是为什么呢？如果评论了，他们又是以何种方式来评论的？调整你所做出的改变，然后再尝试一次。

43 建立关系

我在阿肯色州做年轻的爵士萨克斯手的时候，经常有人会问我："你认识 Chris 吗？"我不认识他。显然，Chris 也是一名阿肯色州的高中生，并且是一位有理想的爵士乐演奏者。所以，人们碰到我，就会把我和他联系起来，希望我们相互熟识。

一年夏天，我有机会去看贝西伯爵爵士大乐团在阿肯色河畔一个圆形露天剧场的表演。在良好的氛围和异常勇气的共同作用下，我来到后台，在乐手们离开之前与他们攀谈起来。我不是个健谈的人，所以这次真的是命运的安排。我站在后面与其中一名萨克斯手交谈时，一个男孩走了过来也攀谈起来。5 分钟或者 10 分钟后，乐队离开了，留下我们两个年轻人各自站在那里。突然间，我们同时问道："你是 Chris/Chad 吗？"

接下来的几年，我的大部分业余时间都是和 Chris 一起度过的。很快我就发现，Chris 非常善于同这个城市里最好的演奏者交往。他只是一个高中生，却已经开始有演出的机会了，并且是替代 Little Rock 中最有名望的爵士钢琴手。在他这个年纪来说，Chris 相当不错，但是，并不是那么好。

很快我就知道这其中的原因了。每周都会有几个晚上我们一起去爵士乐酒吧看演出。对我来说，有时会感到很不舒服。因为我性格内向，而每当台上演奏的乐队休息的时候，Chris 都会停下和我说了一半的话，然后走过去与乐队成员交谈，把我一个人留在座位上。这样的行为就像钟表一样有规律，Chris 就像一个机器人。我得承认，他这样做，让我觉得有些恶心，他是如此地老套乏味。可怜的演奏者正在休息啊，Chris 这样做，他们不会觉得厌恶吗？他们根本就不想和这个可恶的小孩交谈！而和他一起来的我，不得不跟着他一起走过去，否则我就得一个人尴尬地坐在座位上等他回来。偶尔有

几次我非常累的时候，会选择后者。但是，大多数情况下我还是会和 Chris 一起走过去，努力假装自己融入进了他们的谈话。

但令我吃惊的是，那些演奏者通常都很喜欢和 Chris 交谈——甚至也愿意和我聊天。Chris 非常主动，他会问他能不能和乐队坐在一起，但我觉得这样做是非常不合适的。他还会要求乐手给他上课，这样他就可以去他们的家里，听音乐并和他们谈论爵士乐即兴演奏。偶尔，他也会拉着我一起去，我的感觉也是非常的被动。

对 Chris 与这些乐手建立起来的关系我非常纳闷。他抓住机会，与那些优秀的乐手一起演奏，而我只是和他一起出去玩的某个人。他是我与这个城市中最出色的乐手之间的桥梁。而我们之间唯一的不同之处就是他的性格比我更外向开朗。

多年来，Chris 一直坚持“做最差的”这个战略，同时又毫无顾忌地与乐手们攀谈，这使他成为了不可思议的钢琴演奏者。事实上，他挤进了那些非常著名的爵士乐乐手中间，与他们共同演奏；而我，还是他原来认识的那个我。他把我拉进那些知名度高的爵士乐演奏会中，但是，从来都是他拉我进去——没有一次是我拉他进去。

> **恐惧感使我们无法接近专业人士。**
>
> ***Fear gets between us and the pros.***

那以后，我在不同场合都碰到过这样的人，古典音乐乐手中、软件开发行业里，甚至是在一间小小的办公室里都有这样的人存在。Chris 将其称之为“建立关系”，这让我更加反感。但事实是，非常出色的人是不会介意有人想要认识他们的。人们喜欢被别人欣赏的感觉，而且他们也愿意谈论他们所热衷的话题。没错，他们是专业人士、大师、领军人物或者是著名的作者，但是他们首先是人，人是社会动物，喜欢与人交流。

根据我的个人经验，我认为我们这些凡人和那些我们所仰慕的人之间最大的障碍就是我们自己的恐惧。与那些聪明、人际关系好、能教你东西或可以帮助你找到工作的人结识，是取得进步的最好的方法，但是我们却没有胆量尝试与他们结识。音乐家、美术家和其他艺术家通过结成联系紧密的团体，才能够保持出色，他们的艺术作品才得以流传。在社会和职业关系网中，权威就是“超节点”。创建这个关系脉络所需要的就是要少一些谦卑。

当然，你不会随便和任何人建立起这种联系。你肯定想要找到那些和你有共同之处的人。你可能读过某人写的颇具影响力的文章，你可以把你做的工作拿给他，听听他的建议。或者，你为某人创建的系统添加了一个软件接口。这都是合理的与他人建立起联系的好方法。

你可以面对面地建立这种联系，也可以在互联网上建立。持久的联系就是持久的联系。软件这行的高手分布在全球各地。音乐这行也一样，但是你不能想当然地认为音乐家都是通过邮件相互联系的。在音乐世界里，音乐家倾向于按地域形成专业圈子，而软件开发师们的优势是不论我们身处何方都可以很容易地相互取得联系。所以，不仅与当地的专家取得联系非常简单，与其他地方的专家进行沟通也是非常容易的。很多软件开发行业里最具影响力的想法都可以通过邮件甚至是线上即时交谈获得。

我之所以撰写此书，其实是从一封写给 Ruby 的一个发行商的邮件开始的。这封邮件谈论的主题是关于一个 Ruby 的库，以及很多线上聊天记录。尽管我没有勇气把原始邮件发送出去，但显然 Dave 没有觉得烦恼，而现在，他正在读着我写的文字。谢谢你，Chris。

练 习

(1) 挑选出你最喜欢的软件，并给它的开发者写一封邮件。邮件一开始先要感谢他开发了这个软件，接着提出建议、问题或者其他可以与他建立联

系的尝试。请他对你的邮件做出回复。如果这一软件是免费或者是开源的，主动提议来帮忙。

(2) 在你生活的城市里，找一位你敬仰的或者愿意向他学习的人。找一个可以碰到这个人的机会（用户小组会议或者演讲会上都有很大的可能性），主动开始与他交谈，即使这样做会让你感到不舒服。事实上，正是因为这样做会让你感到不舒服，所以你才要迈出这一步。

我们就不能……

——Stephen Akers

Genscape公司，信息技术副总裁

常在办公室的人都知道，IT和商业经营（IT之外的）之间的斗争是永远不会停息的。这种斗争的根源是误解、错误的联系和错误的经营预期。这一矛盾被双方每日使用的各种各样的习惯用语所激化。

IT这行的人最讨厌听到的习语就是“难道我们就不能……”这个习语后面接着的话经常是“我们就不能把工作外包出去么？我们就不能再多派几个人手到这个项目来吗？我们就不能让应用程序快点嘛？我们就不能创建一个新的数据库吗？”

问题在于，很多IT技术人员听到这个短语时，关注的都是“就”这个字，这让他们觉得管理人员认为他们的要求都是显而易见的，并且都是小事，很容易就能达成。如果技术人员没能成功完成这些工作，就证明他们连最简单的事情都做不了，应该找人来替代。

结果，对于此类要求的回答通常都是“不”。技术人员想让管理人员知道他们提出的要求不仅非常复杂且操作难度大，而且这些要求并不是明智的。最终，管理人员会感觉技术人员总是拒绝他们的要求，只得离开，而软件开发人员会觉得管理人员对要做的事情根本就没有一点儿最基本的概念。

我过去就是这么想的。那时我认为，管理层真正需要的人是知道团队成员在做什么的人。这也是为什么，在我的职业道路上，

我选择离开 IT 技术工作而加入管理人员的队伍。我自信满满地期望我们团队的所有项目都能大获全胜，因为我知道什么才是正确的方法。

可笑的是，计划的结果往往不如人所愿。尽管作为管理人员，我取得了成功，但我所经历的却并非如我原来设想的那般壮观。

相反，我发现有很多东西是自己需要学习的。

(1) 某些商业因素几乎对每个项目都会起到抑制作用。由于这些制约因素的存在,有时候你只得实施不那么完美的技术解决方案。

(2) 管理层并不是主观地随意制定项目时间表。很多情况下，解决方案的提交时间会对整个项目乃至整个公司起到连锁反应。

学到这些东西后，我发现原来 IT 技术人员错误地理解了“难道我们就不能……”。这个习语的关键词不应该是“就”，而是“我们”，这个词意味着管理人员把技术人员当做整个团队中至关重要的一部分。他们找到技术人员寻求帮助，共同制定解决方案，这将会迎来整个公司的成功。

所以，当你再听到以前害怕听到的那句话时，忍住要说“不”的冲动。把注意力放在“我们”这个词上，自信地说：“好的，我们可以再分配多一些人手在这个项目上，但是这并不是个好建议，因为……”。只是解释你的难处是不够的，你应该更深地探索管理人员这样做是由哪个商业因素造成的。久而久之，你会学到更多这个商业领域的知识，也会更好地分析、判断待解决的难题。这一能力再加上你的专业技术，就会使你从一个总是说“不”的人变成一个公司管理人员不可或缺的合作伙伴。

第 5 章

保持技术领先



还记得 20 世纪 80 年代的流行歌手 Tiffany（没有姓氏）吗？那时候，她是 Top Forty 排行榜中的歌手，收音机里总是会响起她的歌声。她享受着成功的巨大喜悦，她的名字曾一度家喻户晓。

你最后一次听到她的名字是什么时候？我猜你一定不记得了。反正我是想不起来了。

Tiffany 在 80 年代的时候红极一时——虽然时间不长。随着 90 年代的到来，她就过时了。显然，她累了，没能跟上时代的脚步，抓住歌迷的热情——甚至连“粉丝”的注意力也抓不住了。当人们不再热衷流行音乐，转而迷恋摇滚的时候，一夜之间，Tiffany 就过时了。

你的职业也可能出现相同的情况。这本书描述的过程是一个循环，直到你退休这个循环才会结束。研究、投资、执行、市场，然后重复。在任何一个环节上花费过多的时间，都会有突然间就过时的危险。

如果你不随时保持警惕，这种危险就会悄然而至，给你来个措手不及。当你发现的时候，已经晚了。Tiffany 不会知道摇滚乐将要在音乐圈里大行其道。她全心全意地让自己成为一名青少年热爱的流行音乐巨星。当摇滚乐出现在 Top Forty 中时，她突然之间就过时了，毫无逆转余地。

本章将告诉你如何避免昙花一现。

44 已经过时的技术

大多数人进入 IT 这个行业，都是因为这个行业中总是充满了变化，工作环境是令人兴奋的，总有新的东西要学习。但是，这行中也有一个令人沮丧的现实——我们辛辛苦苦学到的技术知识比一辆新的雪佛兰轿车贬值得速度还要快。今天热门的技术到明天就有可能成为过时的垃圾了。

你引以为傲的新技术已经过时了。

Your shiny new skills are already obsolete.

在 *Leading the Revolution*[Ham02]一书中，Gary Hamel 谈论了各领域的现任领导是如何开始自满，进而变得无知的。你的事业越是成功，你就越有可能习惯企业的现有模式，这就会使你变得极度脆弱，抵挡不住提出新观点的人——即使是一个非常愚蠢的观点，都有可能使你成功的企业模式看上去非常陈旧，就像是你穿了一件又旧又破的外衣出现在迪斯科舞厅里。选择技术也是一样。如果你已经掌握了当前某个热门、重要的技术，比如在这本书出版之际，你掌握了 J2EE 或者.NET，你会觉得非常舒服。掌握这些技术是有利可图的，对吗？任何一个招聘网站或者报纸上的招聘广告都在证明你的决定是对的。

当心！成功使人骄傲，骄傲会使人自满。像 J2EE 这样的潮流看似永远不会过时，但是任何浪潮都是要不就消失，要不最终被后浪推到沙滩上。长时间习惯于现有的潮流会让你毫无防御能力，完全不会去思考如果在没有 J2EE 的世界里，你又能做些什么。

几十年来大家一直都在宣传 COBOL 已经不复存在了。每个新的技术都会被称作“21 世纪的 COBOL”。现在，这个标签被贴在了 Java 身上。尽管我极度讨厌触碰、看见 COBOL，甚至连离它近点也不愿意，我还是要说，称 Java 为 21 世纪的 COBOL 是对它的恭维。尽管很多人都想看 COBOL 消

失，但是它还在这里，并且已经工作了很长时间了。COBOL 程序员整个职业生涯中都在使用 COBOL。但在当今的经济环境下，很难说这种类型的投资是否依然能够奏效。

COBOL 是一个例外——不是规则。很少能有像 COBOL 这样的技术，让技术人员永久地保住工作。我想要说的不是让你放弃自己的主流知识，那样做是不负责任的。我的意思是，你掌握的知识越是主流，过时的危险也就越大。

我们都知道摩尔定律的推断，计算机的性能每隔 18 个月提高一倍。不管这个数字是否精准，我们很容易可以看出现在的技术还是基本按照 1965 年摩尔做出这一断言时候的速度在发展。而且，随着硬件的不断进步，软件技术也得到了迅猛发展。

计算机性能提高一倍。随着技术更新换代的速度越来越快，任何人都无法跟上变化的速度。即使你掌握的技术是全新的，如果你没有开始学习下一个热门技术，那已经晚了。你可以领先于当今的潮流，却处在下一个潮流之后。在这样的环境下，时间是非常重要的。

必须要认识到，即使你现在处于当今潮流的尖端，也极有可能已经在下一个潮流之后了。时间就是一切，学习之前要先动动脑子。现在看起来不可能的事情，2 年间会发生什么变化呢？如果磁盘空间非常便宜，那有一天会不会免费呢？如果处理器的速度比现在快两倍会怎么样呢？哪些是我们不用担心的优化问题？这些发展进步会对热门技术造成哪些改变呢？

没错，这是有点像赌博。但是，如果你不参加，就一定会输。如果你参加了这个游戏，最差的结果也不过是你学习的某种技术在 2 年内都用不上的。所以，你还可以向前看，继续做这样的赌博。最好的结果就是你仍然领先于这个潮流，继续做最先进技术的专家。

向前看，清楚地知道你的技术发展方向，是盲目和有远见之间的区别。

练 习

每周找出时间来研究尖端技术。每周至少找出2个小时的时间来研究新科技，学习相关技术，并动手尝试。制作简单的应用程序。将你正在以当前技术做的项目，用新技术来做出新的原型版本，来理解它们的不同之处，以及新技术能够做些什么。在你的日程安排中加入做这项工作的时间，一定要按时完成。

45 你已经失去工作了

你已经失去工作了。可能每个月你还是领到了薪水；可能你依然在为你的公司创造价值；可能你还是可以使你的老板非常高兴。但是，你已经失去工作了。

> **你不是你的工作。**
> **You are not your job.**

有一件事是可以确定的，那就是任何事情都是处于变化之中的。经济形态也在不断转型中。工作机会涌向国外又转回我们这里。企业家们正在设法去适应这些变化。我们这个行业仍未稳定，就像是一个处于青春期的孩子——有些尴尬、难看，一年一个样——每天都会有新的变化。

所以，如果你的工作是编程，别把自己当成一名程序员。继续工作，但是不要满足于你的工作。永远不要把自己的身份定位于程序员，或者设计师，或者测试员。

其实，把自己与现在的工作过于紧密地联系起来已经不安全了（也从未安全过）。如果你周围的环境处于不断变化之中，工作背景也是不断地改变，那么把自己紧紧依附在工作上会使你与周围的环境不和谐，进而影响你的工作。

在你丢掉工作之前，你可能已经做出了计划，为自己在公司里的职业发展做好了打算。从设计师到架构师，到分析师，然后再到团队领导。

但是，你现在已经丢掉了工作，所有的计划都得改变，计划总是赶不上变化。有理想是好事，但不要对遥远的未来抱有太大的希望。如果你想要射中移动的物体，就不能瞄准这个物体本身，应该瞄准这个物体要移动到的位置。这个过程的路线不是一条直线，最好的情况是一条弧线，但大多数情况下它是一条曲线。

练 习

如果你是一名程序员，尝试以一名测试员或者项目经理的身份来做一两天你的工作。 有哪些不同的角色是你从没考虑过要尝试的。把这些角色罗列出来，并尝试每天以其中一种身份来工作，看看自己是否适合。你可能看不出工作结果有什么不同，但是你会发现你工作的方式发生了变化。

46 没有终点的道路

美国社会存在的一个严重问题就是它是一个以目的为导向的社会。大到学习、职业发展的过程，小到开车的旅途，人们关注的总是结果。我们过于关注事件的结果，却忘记了全局。

仔细思考一下，在逻辑上我们应该要花时间关注的恰恰是结果的反面。基本上，我们大把的时间都花在了做事情的过程中，而结果基本上不会占用时间。举个例子，当你开发软件的时候，你的时间花费在整个开发的过程中，而不是在开发完成的软件上。

你的职业生涯也是一样的，最重要的部分不是晋升或者加薪，而是向这些发展方向努力工作的过程。或者，更重要的是，是你抛开这一切忘我工作的过程。

如果说这才是你工作生活的核心——真正的工作——那么你已经到达目的地了。你一贯使用的以目的为导向、关注终点的思维方式只会导致从一个目标到下一个目标，永远不会结束。大多数人都没有认识到过程就是终点。

回到软件开发这个例子，人们很容易就会倾心关注自己编写的代码的发布。客户需要一个网页应用程序，你就会专注于完成这个应用程序。但是，一个使用中的系统程序永远都不会有“完成”的一天。如果过多地关注最终产品会导致我们分心，忽略真正可交付的产品——一个新实体的可持续发展。

> 不要关注结果，要关注做事情的过程。
>
> Focus on doing, not on being done.

关注结果会使人忘记应该做好过程。糟糕的过程只能创造出糟糕的产品。产品或许可以实现其最低的要求，但是其内部会是一塌糊涂。你达到了短期的最终目标

——但这并不能使产品可持续发展。

不仅糟糕的过程制造出糟糕的产品，糟糕的产品也会导致糟糕的过程。一旦你生产出的产品内部一塌糊涂，你的工作过程就会围绕着它不断做出调整。产品的破窗会导致整个过程的破窗，这是一个恶性循环。

所以，不要总是不停地问："我们完成了吗？完成了吗？"重要的是你穿越这条道路的过程，而不是这条路的终点。

练 习

Naht Hanh 在 *The Miracle of Mindfulness* [Han99]一书中提出了一条建议：在下次洗碟子的时候，不要只是想着要洗完它们。试着享受整个洗碟子的过程。不要关注于洗完它们，而是要关注"洗"这个过程本身。

洗碟子是一项单调的工作，没人愿意做。软件开发师每天也要经历类似单调沉闷的工作，例如时间追踪和费用报告。下次再从事类似工作的时候，不要焦急地想要赶快做完它，尝试在工作中关注任务本身。

47 给自己做一份蓝图

作为软件开发师，从你做系统的工作经验来看，你知道，当做维护工作的时候，人们很容易就会墨守成规，原地踏步。当你在维护一个其他开发师使用的应用程序或者一个库的时候，除非你有一份功能路线图，否则工作就会在漏洞修复模式停滞不前。可能因为用户的要求或者你自己的需要，你会偶然做一下提升，但是代码通常会停滞在一个稳定的状态，改变的速度非常慢，因为你认为这个系统已经完成了。

但是一个使用中的程序是永远没有完成的一天的，除非它要退休了。你和你的职业也是一样的。除非你不想在这个行业干了，否则你就需要一份蓝图。如果微软认为 Windows 3.1 已经完成了，那么现在我们所有人都只能使用 Macintosh。Apache 的开发师们要是认为他们的网络服务器达到 1.0 时就已经完成了，那么他们现在就不会引领这个市场了。

个人的产品路线蓝图是用来判断你是否在不断向前发展的依据。每天，你出入相同的办公室，从事大量相同的工作，周围的环境没有任何变化。所以，你需要在合理的范围内做出一些标记，当你达到这些标准的时候，你就知道你不是停滞不前的。产品的“功能”就是这些标记。

只有当你真正做出计划之后，才能看到下一个标记。在第 2 章和第 3 章中，你已经知道了如何做出职业选择，以及如何在职业上做出投资。尽管我集中谈论的职业投资选择看似是一次性的，但每一次选择都应该是整体的一部分。把每一种新的知识或者能力看作等同于一个程序中的一种功能。一个仅有一种功能的程序算不上是程序。

另外，如果程序具备许多功能，而这些功能却不具备整体性，那这个应用程序的使用者就会感到十分困惑。这到底是一个地址簿还是聊天程序？它

到底是一个游戏还是一个网络浏览器？一份个人产品蓝图不仅可以帮助你一直处于正轨上，不断向前发展，还能够帮助你纵观全局，告诉你应该要做什么。它告诉你做出的每一次选择都是整个职业发展的一部分。有些选择可以共同作用，推动你的职业发展，但也有些选择缺乏关联性，跳跃性太大。他是一名系统管理员还是一名设计师？她是一名程序架构师还是一名自动化测试专家？

尽管学些不同的技术是绝对正确的——这会拓宽你的思路，但是，想想你的技能包会向自己展示什么样的前途，这个想法也不错。没有蓝图，你的前途就会像杰克·凯鲁亚克（Jack Keronac）的小说一样结构松散，无法形成一个逻辑上相互关联的整体；没有蓝图，你可能会迷失方向。

练 习

做出蓝图之前，先画出你曾经所处的位置，这会对你制作新的蓝图起到鼓励和启发的作用。清楚地列出你职业发展的时间表，标注出你从哪里开始，以及在每一阶段你的技术和工作是什么。注意在哪一个阶段你在持续进步，又在哪一个阶段取得了重大进展。注意你每一次取得重大进步的平均时间是多长。当你展望职业发展的时候，把这份蓝图作为参考。清楚地了解过去取得的进步，可以帮助你制定出更加实际的目标。制作出历史蓝图之后，要不断更新。当你不断向新目标前进的时候，这种方法可以很好地反映你的进步。

48 要注意观察市场变化

把钱投进变化莫测的股票市场，然后置之不理，是非常愚蠢的。就算你做足了研究，慎重地选择了要投资的股票，市场依然具有很强的不确定性。做投资的时候，只投资不管理是行不通的。即使现在股票的价值在增长，但这不意味着明天它就不贬值。

你可能正在错失一个良机。你选择的股票可能非常安全，每年都能增加一成的利润。如果说市场上其他的股票没能产生大于一成的利润，那这个投资就是非常正确的。但是，今天你的主力投资，即使它保持着很好的势头，但明天呢？或许它的表现就比不上其他的股票了。

市场在不断变化着，稍不留意就会赔钱或者错失赚钱的机会。

在知识上的投资也是一样的。现在，Java是保守的选择。但是如果有一天情况改变了呢？你又怎么才能知道情况改变了呢？

举个例子，如果Sun公司开始显现出没落的迹象，那会怎么样呢？近几年，Sun公司已经丧失了市场主导地位，Java不是一个开放标准，尽管现在开源是由Sun公司支配和开发的。垂死挣扎的Sun公司可能会想尽一切办法让它的语言和虚拟机成为最后一搏的利润中心。这可能会使Java语言发生一些不和谐的改变，导致整个行业范围内的恐慌。

但是，当你意识到这些问题的时候可能已经太晚了，因为你在全心全意地潜心研究代码。可能突然间你会发现自己的技术已经不再那么有价值了。这只是一个不太可能发生的虚构事例，但是类似的情况可能会发生。

更加可能发生的是，如果你对当前的技能非常满足，当下一个热门技术

来临的时候，你可能会无视它的出现。10 年前，你可能会吃惊于带有垃圾收集的面向对象的程序语言会发展到如此之大，但是如果你细心观察了，是绝对可以看出迹象的。谁又能预测出 10 年后的大事记会是什么呢？

你必须要时刻细心留意。注意技术方面的新闻——不管是商业方面的还是纯技术方面的。O'Reilly and Associates 公司的 Tim O'Reilly[①]曾说，要留意那些技术达人。技术达人是指那些总是站在技术最前端的人，至少在他们感兴趣的领域中他们总是走在尖端。通过我的观察，Tim 的意思是说如果你能找到这类人，观察他们热衷于什么，那你就能够大致了解到什么技术将会成为热门，或者预测出两年后的热门是什么。这招非常灵。

留意那些技术达人。
Watch the alpha geeks.

不管你选择做什么，必须要清楚地知道在技术领域，无论你今天选择投资的技术是多么得尖端，但注定有一天它会被淘汰。如果你不细心观察市场的变化，那这些变化可能会让你措手不及，让你大吃一惊——你绝对不会想要经历这种感觉。

练 习

明年开始尝试做技术达人，或者至少结识一位技术达人，与他建立紧密的联系。

① http://tim.oreilly.com/。

49 镜子里的胖子

很不幸——我超重，而且已经有很长一段时间了。尽管在印度的时候，体重减轻了一些。部分原因是因为当时饮食比较健康，我也做一些运动，但主要是因为那时候我生病了。回到美国之后，体重又慢慢地长了回来。这让我很是失望，所以我加入了健身俱乐部并找了一名健身教练。体重又开始下降了。

我的体重这样起伏变化过好几次。最神奇的是我无法知道自己什么时候增重了，什么时候减重了，只能通过别人告诉我，或者哪天我的衣服突然间不像以前那么合适了，我才知道体重发生变化了。我妻子每天都能见到我，所以她也看不出我的变化，而且在美国，如果你长胖了，通常别人是不会告诉你的。但是在印度，人们会告诉你，你长胖了。

我不知道自己长胖了还是变瘦了，是因为我天天都能看到自己。如果你每天都看着相同的事情，也同样很难看出它的变化，除非是突然间的改变。就像你坐在那盯着一朵花蕾，是看不出什么变化的。但是，如果你离开两天，然后再回来看，你就会看到非常明显的变化。

职业也是一样。事实上你根本就不去观察注意。这就是问题所在了。每天你都看着相同的自己，一点变化也看不出来。你看起来和以前一样适应那个环境；你看起来和以前一样具有竞争力；你的技术看起来和以前一样处在时代尖端。

但是，突然有一天你的工作（或者你所在的行业）不再适应你了。刚开始会有点不舒服，但是这其实已经是关键时刻了，你要不就赶快做出行动，要不就换个行业。

体重的波动可以通过磅秤来衡量，很容易就能看到你取得的“进步”（或

者对我来说，是“不足”）。遗憾的是，作为软件开发师，没有磅秤可以用来衡量你的市场适应能力或者你的技术。如果有这种磅秤，我们就可以让你坐在磅秤上然后自动生成付给你的薪水了。既然没有这种衡量工具，你就必须要找到衡量的标准。

找到一个可信赖的第三方，是衡量进步的简单方法。一名良师或者一位关系好的同事不是你肚子里的蛔虫，所以可以比较客观地对你做出评价。你可以和他们谈论你作为一名软件开发师、项目领导、沟通者、团队成员等任何与你相关的能力。在通用公司，有一项被称作360度评估的方法，这种方法可以鼓励员工从同事、上司和客户那里得到回馈意见。作为雇员，这种方法可以很好地帮助你从不同的角度来审视自己。

开发员们，要自我反省。
Developer, review thyself.

做此种工作的时候（独自完成或者在别人的帮助下完成），最重要的就是要弄清楚自己的盲点在哪里。只需要知道它们在哪里，而不需要你改正所有的盲点。弄不清楚这点，你对你的盲点还是会视而不见，这样就会对你产生不利的影响。问题总会出现，所以最好要做好准备。

即便你有一个神奇的磅秤可以帮助你测量自己，那前提也是你要使用它。要制定自我反省的时间表。不明确制定出自我反省的时间，你就永远不会去自我反省。仅仅提醒自己“别忘了去拿反馈意见”是不够的。如果你有具备提醒功能的日历，安排一个自我评价的时间。首先要制定你的衡量系统，然后把它提上日程。把这项工作作为你工作生活的一部分，否则你是不会去做的。

如果你的公司已经具有此类评价系统，别把它看成是人力资源部门的无聊行为。应该认真填写。在你的公司里，这个过程或许并没有充分地发挥作用，但是这项工作的动机是正确的。

最后，当你设定好评价系统并且为此项工作安排了确定的时间后，写下此项工作的结果。将这份评价放在手边，经常回顾并更新。将自我评价过程

以文字的方式记录下来会使这个过程更加具体化。

别让过时悄悄地降临到你的身上，就像你的裤子突然间就变得紧身了一样。

练 习

(1) 做一次 360 度评估。

- 想想哪些人是你可以信赖的，并且你可以非常自在地请他们给出对你的评价。列出人名单。这一名单内最好要包括你的同事、客户、上司和下属（如果你有下属）。
- 再列举出你认为作为专业人士需要具备的 10 项重要特征作为衡量标准。
- 将这个列表转化成一份调查问卷。在这份调查问卷上，一定要包含下面这个问题要求参与者按每项特征对你做出评分。最后再加一个问题："我还应该问哪些问题？"
- 将调查问卷发放给你列表中的人。要求他们以批判的角度对你做出建设性的评价。你需要的是诚实的评价——不是糖衣药片。

当你得到回馈之后，仔细阅读每一份调查问卷，并依此制定出下一步行动计划。如果你向正确的人询问了正确的问题，那么你会得到一些能够付诸实践的回馈。与你的评价者共同分享这些结果——不是分享他们给出的答案，而是你根据这些答案做出的行动计划。记得要向他们致谢。定期重复这一过程。

(2) 开始记日志。可以是网络日志，就像我们在第 4 章第 39 节中所讨论的，或者写日记也行。记录你在做什么工作，在学习什么以及你对这个行业的一些观点。

坚持记录一段时间之后，回顾前面的记录，现在你仍然同意当时的想法吗？那些想法现在看起来幼稚吗？你改变了多少？

50 南印度捉猴陷阱

在 *Zen and the Art of Motorcycle Maintenance*[①][pir00]一书中，作者 Robert Pirsig 讲述了一个南印度人如何捉猴子的故事。我不知道这个故事是不是真的，但故事具有启迪作用，下面我将转述此故事。

南印度人多年来一直受到猴子的困扰，他们发明了一种巧妙的方法来捉猴子。他们在地上挖掘一个又长又窄的洞，然后用一个同样长窄的物体来扩宽这个洞的底部。他们把米倒入这个拓宽了一些的洞底。

猴子很贪吃，这也是猴子惹人厌的重要原因。他们会跳上车子或者不顾危险追着一大群人，就从你的手里抢走食物。南印度人一直为此困扰，时刻警惕着这些猴子。（相信我，当你安静地站在公园里，突然间一只猕猴"嗖"地过来抢走你身上的某些东西，这种感觉让人非常忐忑不安，没有安全感。）

所以，Pirsig 说，猴群会发现这些米，将胳膊深深地伸进洞里，手会到达底部。他们会贪婪地抓米，直到手里装不下了为止，这个过程中手会攥成拳头。洞底较大的部分可以容下他们的拳头，但是其他较窄的地方就太窄了，猴子的拳头就卡住了。

当然，猴子们可以放弃食物，获得自由。

但是，猴子非常看重食物，看重到根本无法强迫自己放弃食物。他们会一直抓着那些米直到把米拉出来，丧失生命也不会放弃。基本上它们都会为了食物失去生命。

Pirisig 讲述这个故事是为了阐述一个概念，他称之为价值僵固。当你过于坚信某事的价值时，就会无法客观地来评判它，这即是价值僵固。猴子过

① 中文版名为《万里任禅游》已由重庆出版社出版。——编者注

于看重米的价值，所以它们无法看清放弃米就能得到自由。猴子这样做看起来非常地傻，但是我们每个人都有自己的“米”。

如果有人问你帮助发展中国家资助饥饿的孩子是不是正确的做法，你很可能连想到不会想就会回答“是的”。但是如果有人质疑这个观点，你可能会觉得他们简直是疯了。这就是价值僵固的例子。你过于坚信你的看法是正确的，所以无法想象有人居然会持不同的观点。显然，我们对某事的坚持不一定都是不好的。对大多数人来说，宗教信仰（或是没有宗教信仰）是个人信仰和价值观的一部分，不容质疑。

但不是所有坚信的价值都是正确的。而且很多时候，在某种环境下是正确的事情，到另一种环境下就不一定是正确的。

价值僵固使你脆弱。
Rigid values make you fragile.

举个例子，在做技术选择的时候，我们很容易就会非常烦恼，尤其是当我们选择的技术处于劣势。我们非常热爱这门技术，如此地看重它的价值，当有其他的技术选择时，我们会像打仗一样努力捍卫住它的地位——即使我们拥护的选择明显是错误的。我自己遇到过的类似的例子（自己也很内疚）是对Linux 的过分热衷。很多 Linux 用户都将 Linux 放在每一个前台、助理和公司副总裁的电脑桌面上，但事实是就可用性而言，这套工具是比不过很多商业操作系统中可使用的商业软件的。当你把正确的软件交给错误的人使用时，你会使你的客户不高兴，你自己也会显得非常愚蠢。

每天你都能够看到自己，所以看不出自己的体重发生了变化。价值僵固也是同样的。每天我们都在工作中度过，在做职业选择的时候很容易就会产生价值僵固。我们知道什么奏效，并一直坚持着。或者，可能你一直想被提升到管理岗位，所以你一直向那个目标努力，却忽略了自己是多么热爱编程。

你选择的技术也可能已经过时了，使你突然之间失去了立足的基础。就像一只在慢慢加热的热水里的青蛙，突然就发现自己处于不利之处。很多人

都知道 20 世纪 90 年代中期，Novell 公司的 Net-Ware 操作平台提供文档和打印服务的时候，Novell 公司凭借目录服务产品走在那个时代的尖端，而我们这些“知情者”非常狂妄自信，批判一切与之竞争的技术。Novell 公司的产品在当时享有绝对的市场占有率，根本无法想象有一天局势会改变。

在没有任何一个事件作为明显标志的情况下，Novell 公司败给了微软。微软公司从没有发布过神奇的目录服务产品让我们大吃一惊，“哇！不用 NetWare 了！”但是，Netware 就是慢慢地从时代尖端的创新者没落成技术遗产了。对很多 NetWare 的管理者来说，在他们还没感觉到壶里的水温升高的时候，水就突然沸腾了。

无论是你职业的发展方向还是你拥护并投资的技术，都要小心那个“捉猴陷阱”。之前认真做出的选择可能就是你职业走向没落之前你抓到的那最后一捧米。

练 习

(1) 找到你的捉猴陷阱——你的价值僵固是什么？在你完全不知情的情况下，哪些价值观在引领着你日常的行动。

制作一个表格，划分出“职业”和“技术”两栏。在每一栏中列出你认为是不容置疑的价值。举个例子，在“职业”一栏中，你一直认为什么是自己的强项或者弱项？你的职业目标是什么（“我想成为一名执行总裁！”）？实现目标的正确做法是什么？

在“技术”一栏中，列出你选择投资的技术中，哪些是你最看重的。做选择的时候，你认为什么技术特性是最重要的？你是如何制作可扩充的系统的？开发软件最具生产力的环境是什么？总体上来讲，最好和最坏的平台是什么？

当完成这个列表之后，进行反向思考。如果你认定的事实的相反方面是正确的呢？请你诚实地向你认定的每一项事实发起挑战。

这就是你自己弱项的列表。

(2) 了解你的敌人——找出你最讨厌的技术，并用它来完成一个项目。开发员喜欢将自己划分在某个竞争阵营中。.NET 开发员讨厌 J2EE，J2EE 开发员讨厌.NET。UNIX 开发员讨厌 Windows，而用 Windows 的开发员又不喜欢 UNIX。

选择一个简单的项目，尝试用你讨厌的技术制作一个出色的应用程序。如果你是 Java 工程师，那就让那些.NET 开发员看看一个真正的开发员是如何使用他们的平台的！这样做最好的结果是，你发现一直以来你非常讨厌的技术其实也不是那么糟糕，而且用它还能开发出不错的程序。你还能掌握（当然，不是那么熟练）一种新的技术，以后或许会成为你的优势。最坏的结果是，做这个项目让你实践了一把，你也有了更好的证据来捍卫你的观点。

51 避免瀑布型职业计划

本世纪初，软件业出现了一个小动荡。那时在软件这个行业中，失败远远多于成功，业界的一些专家认为他们可以转变这个局势。他们称自己为“敏捷联盟”。

那时这个行业相信唯一正确的软件开发方法是遵循自上而下、缜密设计的严格过程。分析师规定功能要求，架构师制定好结构，再由设计者制定出详细的设计方案。开发师拿到设计方案后用某种编程语言将其编写成代码。历经几个月——有时候是几年时间的努力后，这些代码会形成一个整体然后送到测试团队的手中进行测试证明。

有时候这一过程也可以有所改变。如果每个人在项目一开始就清楚他们需要做的每一个细节，那么这种计划性和精确性会生产出缜密并且品质受控制的软件。但是大多数情况下，人们并不知道一个大项目中的每一个细节。项目越大细节越复杂，就越难想象出制定规范的每个功能的细节。这一过程就是“瀑布过程”，等同于今天我们所说的糟糕的过程。

所以，就像“敏捷联盟”成员所发现的，按照这种繁琐的程序工作会生产出经过严格测试、文件齐全的软件，但是它并不是软件使用者想要的。敏捷联盟想要创造出一系列敏捷方法，使软件开发过程向简单的方向改变。缩短花费在计划和设计上的时间。软件具有延展性，所以改变开发程序也并不会造成很大的开销。敏捷开发方式主张改变应该成为软件开发的一部分，且以节省开支和提高效率为基准，随时进行调整改变。

现在，这一切听起来是显而易见的。但是在那个时候，敏捷编程开发方式引起了分歧和辩论。理论上讲，详细计划和严格执行显然是正确的；但在实践上，这一理论是行不通的。

以我最初采用敏捷开发方法（特别是极限编程）的经历来看，通过敏捷开发方法，我看到了以前看不到的一切。动机和影响力不仅限于软件开发，而是变得更具广泛性。每当我遇到复杂的问题时，以一种容许反复改变的方式来解决问题会减轻压力，使我的工作更具效率。

也不知道为什么，我用了很长时间才认识到我遇到过的最复杂的项目——给我最大压力也是最重要的项目——就是我自己的职业。一直以来，我都是以瀑布开发方式来设计自己的职业发展。这种开发方式不仅给软件开发项目制造了问题，也开始对我和我的职业产生了影响。

那时我正在走向成为一名成功的公司副总裁或者信息主管的道路上，在这条道路上我的表现不错。我很快就从一名菜鸟开发员变成了一名软件架构师，到经理再到总监，并且在这条道路上不断晋升。但是，尽管我如此地成功，我开始觉得我所做的这一切并不是我喜欢的。事实上，我越是成功，就越不可能做我喜欢做的工作。

这和繁琐严格的程序对他们的客户产生的影响是一样的。我出色地工作着，却为自己创造着我并不想要的职业道路。

我一开始并没有认识到，解决这个问题的方法很简单——改变职业道路。不同的人做出的改变不一样。对我来说，就是回到最初使我开心的信息技术领域。对其他人来说，可能意味着从负责系统管理转变成从事软件开发，从非 IT 领域进入计算机编程领域，或者干脆放弃自己的专业从事其他真正热爱的事情。

就像软件开发一样，这种改变的难度不会很高。当然，从软件测试员转变成一名律师是很困难的，但是从管理岗位转向编程并不是很难，反之亦然。换一家新的公司工作或者换个新的城市工作，也不难。

这种改变不是说让你去建造一幢摩天大楼，需要你放弃之前的一切。现在我每天用 Ruby 编程，我当经理或者在外包开发团队工作的经验与此相关并且有助于我现在的工作。我的雇主和客户了解这一点，并且充分利用我的这一优势。

非常重要的一点是，职业的改变不仅是有可能的，并且是非常必要的。作为软件开发人员，你绝不会想全身心地开发某种客户并不想要的软件。敏捷开发方法帮助你避免此种问题的产生。你的职业道路也是一样。树立远大的目标，但是要在实现目标的道路上，根据情况不断进行更正。从实践中学习，不断改变你的目标。最终，我们都想让客户满意（特别是当制定自己的职业规划时，我们就是自己的客户）。

52 每天都有进步

修补一个漏洞（通常）很简单。你知道某种东西坏了，是因为有人向你汇报。如果你能复制这个漏洞，那么修补这个漏洞就意味着更正引起它的所有故障并确认无法再复制这一漏洞。要是所有问题都这么简单，那该多好！

但是，并不是所有的问题和挑战都是如此。大多数重大的挑战总是在不可逾越的潜在失败中显现出来。软件开发、职业管理甚至生活方式和健康都是如此。

一个复杂且充满漏洞的系统需要一次详细的检查。你的职业发展停滞不前。你的生活方式就是整日坐在那里伏案编程，身体也变得一团糟。这些问题都比修复一个漏洞要严重得多。这些问题都非常复杂，很难衡量，并且由很多小的解决方法组成——某些方法根本无法奏效。

正是因为问题的复杂性，在重大的问题面前，我们很容易就失去了动力，转而去关注那些容易衡量和解决的问题。这就是为什么我们经常延误了问题的解决时间，而这种拖延又导致了我们的内疚，这让我们感觉糟透了，结果就是再继续拖延。

在第 49 节中我提到长期以来我一直努力减肥和保持身材。事实上，如果你严重超重，就很难对“恢复身材”有什么概念，更不用说做出什么具体行动。更困难的是，如果你为了改善这种情况做了些努力，当时是看不出效果的，甚至一个星期之后你还是看不出任何改变。事实上，你可能整天都在健身减肥，但是一个星期之后，看不到任何成果。

这就是导致你在问题面前丧失动力的因素，使你连尝试都没尝试就投降了。

最近我正在认真地解决这个问题。几乎每天都去健身房，饮食也更加健康。但是即使当我非常认真地对待这项工作的时候，也一样很难看到结果。

最近的一个晚上，当我沉溺于失去动力的时候，我们的朋友 Erik Kastner 在社交网络 Twitter 上发布了一条信息：

帮助我减肥……每天问我一次“今天比昨天有进步了吗？（饮食/锻炼）”——今天的答案是：是的！

当我看到这条信息的时候，我知道这就是恢复健康的方法。秘密就在于无论你要改进的是什么，注意今天与昨天相比，是不是取得了进步。就是这样。这很容易。就像 Erik 一样，这样就能够充满激情地向着遥远的目标迈出真实的看得见的脚步。

最近，我还在解决我所见过的 Ruby and Rails 中一个最复杂难搞的程序。这份工作是我们公司从另一个开发员那里接替过来的，作为一个咨询项目。这一程序中有一些需要实现的主要功能，和很多需要更正的漏洞和性能。当我们打开防护罩想要工作的时候，发现里面一团糟。那家公司给我们的时间和资金都很有限，所以就算这是你扔掉的代码，我们也没时间重新来过。

所以我们艰难地进行一个个小的修复，每一个都要花费比预期长的时间。开始的时候，这就像一个巨大的怪物，永远不会有完结的一天，我们非常疲惫，工作毫无兴趣可言。但是慢慢地，修复的速度变快了，之前不可接受的性能也被改善了。这是因为我们决定让代码每天都比前一天有些改善。有时候我们会将一套很长的解决方案分解成几个小的方法。有时候会将不属于这个模型中的继承体系移除掉，或者修复一个失效很久的单元测试。

当我们逐渐地做出这些改变后，问题就迎刃而解了。重构一套解决方案占用的不过是你喝杯咖啡或者和同事闲聊新闻的时间。一个小小的改变都会起到推动的作用。一旦你做出了这些改变，你就会清楚地看到不同。

每一个改变之后，你可能无法看到整体发生明显的不同。当你努力得到你同事的尊重，或者努力变得健康的时候，每天取得的进步不会直接导致有形的结果。就像我们之前所说的，这是因为远大的目标会使人丧失动力。所以，对于努力实现那些远大又有难度的目标，非常重要的一点就是不要总想着每天都要离最终目标近一些，而是应该想每天要比前一天做得更好。举个例子，我不能保证我今天比昨天更加苗条了，但是我可以控制今天是不是比昨天付出了更多的努力去减重。如果我付出了更多的努力，那我就有权力为我所做出的努力而自豪。通过这种持续可见的进步，我不再感到内疚也不再拖延时间。而内疚和拖延时间正是我们战胜重大困难的因素。

即使是一点儿小进步，你也应该感到高兴。今天比昨天多写了一个测试，这已经帮助你向“更好地做单元测试”迈进了一步。如果你是从头开始，每天多写出一个测试，并保持这个速率，当你发现你无法再超越昨天的时候，就会发现自己现在已经可以“更擅长单元测试”了，也没有必要做相同的改进了。但是，如果你在第一天就想从零开始，一口气写出 50 个测试，那第一天会非常地辛苦，很可能你就不会再坚持下去了。所以，要逐渐慢慢地做出改进，从小做起，但是每天都要坚持。小的改进会降低失败所要付出的代价。如果有一天你没能坚持，那么明天你会有一个新的基线。

最棒的是，这条简单的箴言可以应用于非常具体的目标，比如完成一个项目或者更改软件。它也可以应用于非常高水平的目标。在改善职业发展的道路上，你今天是如何比昨天做得更好的呢？多交了一个朋友，又向一个开源项目提交了一个补丁包，撰写了一篇有深度的博文并将它发布在了你的网络日志上。今天与昨天相比，你又在技术论坛上多为一个人解决了问题。如果在改善自己的道路上，每一天都比昨天多做出一点改变，你就会发现——拥有卓越的职业生涯——这个目标变得越来越容易达成，而不再像汪洋大海一样没有边际。

练 习

列出你想要做的复杂困难的改善——可以是个人问题也可以是职业问题。如果你的列表很长，那也没关系。然后，思考这个列表上的每一项，想想你今天能做点什么可以使你自己或者这项问题比昨天有些改善呢。明天，再看一遍这个列表。昨天和前天相比，你有什么进步吗？怎样做能使今天比昨天更进一步呢？把这项工作安排到你的行程表中。每天早上用两分钟的时间来思考。

53 独立

在困难时期，我总是回想在大公司工作的日子。那时候，我被安置在办公室的格子间里，公司有着繁琐空洞的管理组织架构。在大公司里，一个聪明的员工可以什么都不做，却依然能在公司里生存下去。那时候对我们来说这是个笑话，但却也是事实。大多数情况下，如果项目没有完成，由于中间经手的管理组织架构和人过多，根本无法找到问题到底出在哪里。如果项目完成的时间比应该使用的时间长，是因为根本没人知道任何项目到底应该花费多长时间完成才是正确的，这也是因为繁琐的管理组织架构造成的。

所以，如果有一天你真的累了，大公司允许你坐下来休息休息，上网冲浪或者早点下班回家，甚至“请个病假”。尽管我一直抱怨在大公司里工作多么多么糟糕，但是不得不说，在大公司工作确实也有它的好处。

问题是，尽管公司的层层组织架构降低了运营风险，却也因此降低了工作效率。如果你可以隐藏在平庸的盾牌之后，就会丧失变得卓越的动力。即使是像我们这样的人，也无法抵挡 YouTube 或者我们最爱的网络漫画[①]的诱惑力。

这样看来，如果你筋疲力尽，大公司是个不错的选择。但是如果你在为了成就卓越的自我而奋斗（你正是在这么做！），那么大公司可不是个正确的选择，就像如果你正在努力甩掉腰间的赘肉，那绝对不能去蛋糕店工作。解决方法是什么？学着独立！

你有技术，并且是经过自己不断磨练的技术。你知道自己的价值是什么。成为一名独立承包人是最终考验的一部分。没有任何组织结构可以成为你的庇护。你要直接对付你薪水的人负责。你所做的任何一件事都直接反映在你

① 推荐一个不错的网络漫画网站 http://toothpastefordinner.com，我在这里找到了不少乐子。

的工作业绩中。如果你犯错误了，没有任何人和你一起承担责任。只有你自己，你的专业和你的执行能力。

成为一名独立的承包商还会迫使你学习如何推销自己，同时在你专注的领域和技术中检验你的选择。在大公司里工作的时候，都是别人分配工作给你。但是当你独立承担工作的时候，就不能等着客户主动来找你。你必须要走出去，主动去找客户。一旦你找到了客户，你还必须要说服他们，让他们相信你的价值。

你的价值到底是多少，这也是需要你自己来决定的。你所做的工作值每小时 50 美元？还是 250 美元？你怎样来衡量你的价值？你又真得像你自认为的一样有价值吗？

独立不是件简单的事情。这把你所有的技术作为一个专业来测试。现在你可能还没有做好准备。但你也没有必要做足这一切。把它当做是个人发展项目，在业余时间去推销自己。制定目标，以某种速率与客户签订合同，然后利用晚上或者周末的时间完成项目（但是千万不要利用上班时间做!）。在保留安全感的同时，你可以学到很多东西。最坏的结果不过是有几个星期你得加班加点地超负荷工作，结果却没能成功完成这个项目，然后回到你舒服的格子间里，以一种全新的感觉对你的工作心存感激。最好的结果是，你成功了，热爱这份工作，为自己的职业满足感和荷包充盈找到了一条新的途径。

评论家 Sammy Larbi 建议了另一种学着独立的方法。如果你现在在一家大公司里工作，考虑换个小公司工作。如果你在一家老企业里工作，换个新成立的公司。在刚成立的小公司工作，可以一箭双雕：一份全职带薪工作和直接对工作失败负责的挑战。

好奇是一种优点

——独立咨询师/程序员 Mike Clark

我父母说从小我就是个好奇的孩子。我不停地问这问那，阅读一切可以找到的东西，把东西拆开来看它的内部结构。事实证明，这种好奇并不是阶段性的——我一直都是贪得无厌，对什么都好奇。我相信好奇可以成为一种优点，尽管很多人都忽略这个事实。其实只需要做一点练习，就能发展这一优点。

回想过去，我发现自己几次职业道路上的改变都是出于我的好奇。我希望我的故事激励你，让你可以跟随着你的好奇。

我从没想过有一天会成为一名程序员。我一直幻想着自己做些和飞机或者宇宙飞船相关的事情，所以我就读于安柏瑞德航空大学的航空航天工程系是自然而然的选择。但是经过一年的埋头苦读之后，我发现计算机系的学生们获得了许多乐子。作为新学位课程的一部分，他们把计算机科学应用到了与航天相关的问题上。高中时代我就对计算机很好奇，但却从没想过把编程作为职业。所以，我开始和计算机系的学生们频繁接触，看看他们到底在做些什么。不久，我就更换了专业。这次改变是我做出的正确的选择之一。课程仍然具有挑战性，但是每一分钟都令我非常享受。最初对编程的一点好奇很快就转变成了热情。我申请了在NASA（美国航空航天局）的实习工作，欣然开始了我的编程职业生涯。直到今天，我仍然对同事在做什么可以找乐子感到好奇，也从未低估过此种好奇带给我的潜在收获。

一旦我感到满足的时候，我知道是该尝试点新的东西了。这些年我一直在航空领域开发嵌入式软件，我对 C 和 C++语言驾轻

就熟（对我来说也有点厌倦了）。这次是网络编程激发了我的好奇，主要是因为他和嵌入式系统编程截然不同。可惜，我白天工作时做的项目不能与网络连接（这是一项绝对机密的项目），所以我利用晚上和周末的时间学习编写网络软件，并因此获得了用Java开发新项目的机会。最终，我为许多项目和雇主建立了以网络为基础的Java应用程序。对网络编程开发的好奇是我发展多样化技术的催化剂，进而造就了我职业发展的好机会。

学习Ruby and Rails是我一时心血来潮。Ruby是一种有意思的编程语言，使我从另一个角度看待编程。Rails和网络应用程序一样。那时，我的客户中没有人要求使用Ruby或者Rails工作，但没关系。我很好奇，控制不住自己。我利用按小时付费的工作，潜心钻研Ruby和Rails。没想到2005年初，我就得到了建造最早的Rails商业应用程序的机会，而且还受到Dave Thomas的邀请帮助他解决他的Rails一书中的问题。我对新技术的好奇又一次为我的职业发展造就了良好的机会。

我不只对技术好奇，商业经营也同样能引起我的兴趣。这使我勇于在自己身上下赌注，成为了一名独立的咨询师，并创立了一家培训公司（Pragmatic 工作室）。我对经营小公司的好奇给了我学习新本领的机会：销售、市场、客户支持等等。纵观全局也帮助我成为了一名更好的程序员。

所以，有什么东西是令你真正好奇的呢？试着跟随着自己的兴趣，看看会发生什么？结果或许会让你大吃一惊！

“我却对你们说，工作的时候，你们完成了如大地般深远的梦之一部，他指示你那梦是何时开头，而在你们劳力不息的时候，你确在爱了生命。从工作里爱了生命，就是通彻了生命最深的秘密。”

——纪伯伦《先知》

祝你开心

如果你已经是一名仔细思考过职业方向的软件开发员了，那么恭喜你！你可以认为自己是非常幸运的。在很多文化中，可以考虑自己以何为生是少数人才可以享有的特权。作为软件开发员，你就不用去担心如何付房租和填饱肚子了。

你可能已经选择过很多职业发展道路，但是软件开发这条路非常令人振奋。它富有创造力；它需要深刻地思考。作为回馈，它让你觉得你能做到的事情是别人想都没想过的。我们可能会担心如何向更高的阶段进步，如何制造影响或者得到同行的尊重，但是如果你抛开这一切，会发现我们已经做得非常出色了。

软件开发既具挑战又具回报。它像艺术创作一样富有创造性，却又（不同于艺术创作）可以提供具体可衡量的价值。

软件开发极具乐趣！

最后，就我自己的软件开发职业道路来看，最重要的一点不是你以什么为生或者你得到了什么，重要的是你如何接受这一切。这是一种心理活动。满足，就像我们的职业选择，是应该去追寻并且认真选择决定的。

参考文献

[Bec00] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Reading, MA, 2000.

[Cou96] Douglas Coupland. *Microserfs*. Regan Books, New York, 1996.

[DL99] Tom Demarco and Timothy Lister. *Peopleware: Productive Projects and Teams*. Dorset House, New York, second edition, 1999.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable ObjectOriented Software*. Addison-Wesley, Reading, MA, 1995.

[God03] Seth Godin. *Purple Cow: Transform Your Business by Being Remarkable*. Portfolio, 2003.

[Ham02] Gary Hamel. *Leading the Revolution: How to Thrive in Turbulent Times by Making Innovation a Way of Life*. 2002.

[Han99] Thich Nhat Hanh. *The Miracle of Mindfulness*. Beacon Press, 1999.

[HT00] Andrew Hunt and David Thomas. *The Pragmatic Programmer:From Journeyman to Master*. Addison-Wesley, Reading, MA, 2000.

[Pir00] RobertM. Pirsig. *Zen and the Art of Motorcycle Maintenance:An Inquiry into Values*. Perennial Classics, reprint edition edition, 2000.

[Sil99] Steven A. Silbiger. *The Ten-Day MBA: A Step-By-step Guide To Mastering The Skills Taught In America's Top Business Schools*. Quill, 1999.